INTERNATIONAL STANDARD

ISO/IEC 11889-4

Second edition
2015-12-15

Information technology — Trusted Platform Module Library —

Part 4: Supporting Routines

Technologies de l'information — Bibliothèque de module de plate-forme de confiance —
Partie 4: Routines de support

Reference number
ISO/IEC 11889-4:2015(E)

CONTENTS

Tables

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: Foreword — Supplementary information.

ISO/IEC 11889-4 was prepared by the Trusted Computing Group (TCG) and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

This second edition cancels and replaces the first edition (ISO/IEC 11889-4:2009), which has been technically revised.

ISO/IEC 11889 consists of the following parts, under the general title *Information technology — Trusted Platform Module Library:*

– *Part 1: Architecture*

– *Part 2: Structures*

– *Part 3: Commands*

– *Part 4: Supporting routines*

Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured the ISO and IEC that he/she is willing to negotiate licences either free of charge or under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO and IEC. Information may be obtained from:

Fujitsu Limited **1-1, Kamikodanaka 4-chrome, Nakahara-ku, Kawasaki-shi, Kanagawa, 211-8588 Japan**
Microsoft Corporation **One Microsoft Way, Redmond, WA 98052**
Enterasys Networks, Inc **50 Minuteman Road, US-Andover, MA 01810**
Lenovo **1009 Think Place, US-Morrisville, NC 27560-8496**
Advanced Micro devices, Inc. - AMD **7171 Southwest Parkway, Mailstop B100.3, US-Austin, Texas 78735**
Hewlett-Packard Company **P.O. Box 10490, US-Palo Alto, CA 94303-0969**
Infineon Technologies AG - Neubiberg **Am Campeon 1-12, DE-85579 Neubiberg**
Sun Microsystems Inc. - Menlo Park, CA **10 Network Circle, UMPK10-146, US-Menlo Park, CA 94025**
IBM Corporation **North Castle Drive, US-Armonk, N.Y. 10504**
Intel Corporation **5200 Elam Young Parkway, US-Hillsboro, OR 97123**

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO (www.iso.org/patents) and IEC (http://patents.iec.ch) maintain on-line databases of patents relevant to their standards. Users are encouraged to consult the databases for the most up to date information concerning patents.

Information technology — Trusted Platform Module Library —

Part 4: Supporting routines

1 Scope

This part of ISO/IEC 11889 contains C code that describes the algorithms and methods used by the command code in ISO/IEC 11889-3. The code in this part of ISO/IEC 11889 augments ISO/IEC 11889-2 and ISO/IEC 11889-3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any code in this part of ISO/IEC 11889 may be replaced by code that provides similar results when interfacing to the action code in ISO/IEC 11889-3. The behavior of code in this part of ISO/IEC 11889 that is not included in an annex is *normative*, as observed at the interfaces with ISO/IEC 11889-3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of ISO/IEC 11889 from the provided code.

The code in ISO/IEC 11889-3 and this part of ISO/IEC 11889 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in ISO/IEC 11889-3 would be compliant.

The code in ISO/IEC 11889-3 and this part of ISO/IEC 11889 is not written to meet any particular level of conformance nor does ISO/IEC 11889 require that a TPM meet any particular level of conformance.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 9797-2, *Information technology -- Security techniques -- Message Authentication Codes (MACs) -- Part 2: Mechanisms using a dedicated hash-function*
- ISO/IEC 10116:2006, *Information technology — Security techniques — Modes of operation for an n-bit block cipher*
- ISO/IEC 11889-1, *Information technology — Trusted Platform Module Library — Part 1: Architecture*
- ISO/IEC 11889-2, *Information technology — Trusted Platform Module Library — Part 2: Structures*
- ISO/IEC 11889-3, *Information technology — Trusted Platform Module Library — Part 3: Commands*

3 Terms and definitions

For the purposes of this part of ISO/IEC 11889, the terms and definitions given in ISO/IEC 11889-1 apply.

4 Symbols and abbreviated terms

For the purposes of this part of ISO/IEC 11889, the symbols and abbreviated terms given in ISO/IEC 11889-1 apply.

5 Automation

5.1 Introduction

ISO/IEC 11889-2 and ISO/IEC 11889-3 are constructed so that they can be processed by an automated parser.

EXAMPLE 1 ISO/IEC 11889-2 can be processed to generate header file contents such as structures, typedefs, and enums.

EXAMPLE 2 ISO/IEC 11889-3 can be processed to generate command and response marshaling and unmarshaling code.

5.2 Configuration Parser

The tables in the ISO/IEC 11889-2 Annexes are constructed so that they can be processed by a program. The program that processes these tables in the ISO/IEC 11889-2 Annexes is called "The ISO/IEC 11889-2 Configuration Parser."

The tables in the ISO/IEC 11889-2 Annexes determine the configuration of a TPM implementation. These tables may be modified by an implementer to describe the algorithms and commands to be executed in by a specific implementation as well as to set implementation limits such as the number of PCR, sizes of buffers, etc.

The ISO/IEC 11889-2 Configuration Parser produces a set of structures and definitions that are used by the ISO/IEC 11889-2 Structure Parser.

5.3 Structure Parser

5.3.1 Introduction

The program that processes the tables in ISO/IEC 11889-2 (other than the table in the annexes) is called "The ISO/IEC 11889-2 Structure Parser."

NOTE A Perl script was used to parse the tables in ISO/IEC 11889-2 to produce the header files and unmarshaling code in for the reference implementation.

The ISO/IEC 11889-2 Structure Parser takes as input the files produced by the ISO/IEC 11889-2 Configuration Parser and ISO/IEC 11889-2. The ISO/IEC 11889-2 Structure Parser will generate all of the C structure constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM, and marshaling code for structures passed from the TPM.

The unmarshaling code produced by the parser uses the prototypes defined below. The unmarshaling code will perform validations of the data to ensure that it is compliant with the limitations on the data imposed by the structure definition and use the response code provided in the table if not.

EXAMPLE The definition for a TPMI_RH_PROVISION indicates that the primitive data type is a TPM_HANDLE and the only allowed values are TPM_RH_OWNER and TPM_RH_PLATFORM. The definition also indicates that the TPM shall indicate TPM_RC_HANDLE if the input value is not none of these values. The unmarshaling code will validate that the input value has one of those allowed values and return TPM_RC_HANDLE if not.

The clauses below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the ISO/IEC 11889-2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

5.3.2 Unmarshaling Code Prototype

5.3.2.1 Simple Types and Structures

The general form for the unmarshaling code for a simple type or a structure is:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**

When the data is successfully unmarshaled, the called routine will return TPM_RC_SUCCESS. Otherwise, it will return a Format-One response code (see ISO/IEC 11889-2).

If the data is successfully unmarshaled, ***buffer** is advanced point to the first octet of the next parameter in the input buffer and **size** is reduced by the number of octets removed from the buffer.

When the data type is a simple type, the parser will generate code that will unmarshal the underlying type and then perform checks on the type as indicated by the type definition.

When the data type is a structure, the parser will generate code that unmarshals each of the structure elements in turn and performs any additional parameter checks as indicated by the data type.

5.3.2.2 Union Types

When a union is defined, an extra parameter is defined for the unmarshaling code. This parameter is the selector for the type. The unmarshaling code for the union will unmarshal the type indicated by the selector.

The function prototype for a union has the form:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, UINT32 selector);
```

where:

TYPE	name of the union type or structure
***target**	location in the TPM memory into which the data from ****buffer** is placed
****buffer**	location in input buffer containing the most significant octet (MSO) of ***target**
***size**	number of octets remaining in ****buffer**
selector	union selector that determines what will be unmarshaled into ***target**

5.3.2.3 Null Types

In some cases, the structure definition allows an optional "null" value. The "null" value allows the use of the same C type for the entity even though it does not always have the same members.

EXAMPLE The TPMI_ALG_HASH data type is used in many places.

In some cases, TPM_ALG_NULL is permitted and in some cases it is not. If two different data types had to be defined, the interfaces and code would become more complex because of the number of cast operations that would be necessary. Rather than encumber the code, the "null" value is defined and the unmarshaling code is given a flag to indicate if this instance of the type accepts the "null" parameter or not. When the data type has a "null" value, the function prototype is

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, bool flag);
```

The parser detects when the type allows a "null" value and will always include **flag** in any call to unmarshal that type.

5.3.2.4 Arrays

Any data type may be included in an array. The function prototype use to unmarshal an array for a **TYPE** is

```
TPM_RC TYPE_Array_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size,INT32 count);
```

The generated code for an array uses a **count-**limited loop within which it calls the unmarshaling code for **TYPE**.

5.3.3 Marshaling Code Function Prototypes

5.3.3.1 Simple Types and Structures

The general form for the unmarshaling code for a simple type or a structure is:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
***source**	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
****buffer**	location in the output buffer where the first octet of the **TYPE** is to be placed
***size**	number of octets remaining in ****buffer**. If **size** is a NULL pointer, then no data is marshaled and the routine will compute the size of the memory required to marshal the indicated type

When the data is successfully marshaled, the called routine will return the number of octets marshaled into ****buffer.**

If the data is successfully marshaled, ***buffer** is advanced point to the first octet of the next location in the output buffer and ***size** is reduced by the number of octets placed in the buffer.

When the data type is a simple type, the parser will generate code that will marshal the underlying type. The presumption is that the TPM internal structures are consistent and correct so the marshaling code does not validate that the data placed in the buffer has a permissible value.

When the data type is a structure, the parser will generate code that marshals each of the structure elements in turn.

5.3.3.2 Union Types

An extra parameter is defined for the marshaling function of a union. This parameter is the selector for the type. The marshaling code for the union will marshal the type indicated by the selector.

The function prototype for a union has the form:

```
UINT16 TYPE_Marshal(TYPE *target, BYTE **buffer, INT32 *size, UINT32 selector);
```

The parameters have a similar meaning as those in 5.3.2.2 but the data movement is from **source** to **buffer**.

5.3.3.3 Arrays

Any type may be included in an array. The function prototype use to unmarshal an array is:

```
UINT16 TYPE_Array_Marshal(TYPE *source, BYTE **buffer, INT32 *size, INT32 count);
```

The generated code for an array uses a **count-**limited loop within which it calls the marshaling code for **TYPE**.

5.4 Command Parser

The program that processes the tables in ISO/IEC 11889-3 is called "The ISO/IEC 11889-3 Command Parser."

The ISO/IEC 11889-3 Command Parser takes as input ISO/IEC 11889-3 and some configuration files produced by the ISO/IEC 11889-2 Configuration Parser. This parser uses the contents of the command and response tables in ISO/IEC 11889-3 to produce unmarshaling code for the command and the marshaling code for the response. Additionally, this parser produces support routines that are used to check that the proper number of authorization values of the proper type have been provided. These support routines are called by the functions in this part of ISO/IEC 11889.

5.5 Portability

Where reasonable, the code is written to be portable. There are a few known cases where the code is not portable. Specifically, the handling of bit fields will not always be portable. The bit fields are marshaled and unmarshaled as a simple element of the underlying type.

EXAMPLE A TPMA_SESSION is defined as a bit field in an octet (BYTE). When sent on the interface a TPMA_SESSION will occupy one octet. When unmarshaled, it is unmarshaled as a UINT8. The ramifications of this are that a TPMA_SESSION will occupy the 0^{th} octet of the structure in which it is placed regardless of the size of the structure.

Many compilers will pad a bit field to some "natural" size for the processor, often 4 octets, meaning that `sizeof(TPMA_SESSION)` would return 4 rather than 1 (the canonical size of a TPMA_SESSION).

For a little endian machine, padding of bit fields should have little consequence since the 0^{th} octet always contains the 0^{th} bit of the structure no matter how large the structure. However, for a big endian machine, the 0^{th} bit will be in the highest numbered octet. When unmarshaling a TPMA_SESSION, the current unmarshaling code will place the input octet at the 0^{th} octet of the TPMA_SESSION. Since the 0^{th} octet is most significant octet, this has the effect of shifting all the session attribute bits left by 24 places.

As a consequence, someone implementing on a big endian machine should do one of two things:

a) allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or

b) modify the code that manipulates bit fields that are not defined as being the alignment size of the system.

For many RISC processors, option #2 would be the only choice. This is may not be a terribly daunting task since only two attribute structures are not 32-bits (TPMA_SESSION and TPMA_LOCALITY).

6 Header Files

6.1 Introduction

The files in clause 6 are used to define values that are used in ISO/IEC 11889-3 and this part of ISO/IEC 11889 and are not confined to a single module.

6.2 BaseTypes.h

```
#ifndef _BASETYPES_H
#define _BASETYPES_H
#include "stdint.h"
```

NULL definition

```
#ifndef         NULL
#define         NULL        (0)
#endif
typedef  uint8_t            UINT8;
typedef  uint8_t            BYTE;
typedef  int8_t             INT8;
typedef  int                BOOL;
typedef  uint16_t           UINT16;
typedef  int16_t            INT16;
typedef  uint32_t           UINT32;
typedef  int32_t            INT32;
typedef  uint64_t           UINT64;
typedef  int64_t            INT64;
typedef struct {
    UINT16        size;
    BYTE          buffer[1];
} TPM2B;
#endif
```


6.3 bits.h

```
#ifndef     _BITS_H
#define     _BITS_H
#define CLEAR_BIT(bit, vector)  BitClear((bit), (BYTE *)&(vector), sizeof(vector))
#define SET_BIT(bit, vector)    BitSet((bit), (BYTE *)&(vector), sizeof(vector))
                                                    \
#define TEST_BIT(bit, vector)   BitIsSet((bit), (BYTE *)&(vector), sizeof(vector))
#endif
```


6.4 bool.h

```
#ifndef     _BOOL_H
#define     _BOOL_H
#if defined(TRUE)
#undef TRUE
#endif
#if defined FALSE
#undef FALSE
#endif
typedef int BOOL;
#define FALSE   ((BOOL)0)
#define TRUE    ((BOOL)1)
#endif
```

6.5 Capabilities.h

This file contains defines for the number of capability values that will fit into the largest data buffer.

These defines are used in various function in the "support" and the "subsystem" code groups. A module that supports a type that is returned by a capability will have a function that returns the capabilities of the type.

EXAMPLE PCR.c contains PCRCapGetHandles() and PCRCapGetProperties().

```
#ifndef     _CAPABILITIES_H
#define     _CAPABILITIES_H
#define    MAX_CAP_DATA            (MAX_CAP_BUFFER-sizeof(TPM_CAP)-sizeof(UINT32))
#define    MAX_CAP_ALGS            (ALG_LAST_VALUE - ALG_FIRST_VALUE + 1)
#define    MAX_CAP_HANDLES         (MAX_CAP_DATA/sizeof(TPM_HANDLE))
#define    MAX_CAP_CC              ((TPM_CC_LAST - TPM_CC_FIRST) + 1)
#define    MAX_TPM_PROPERTIES      (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PROPERTY))
#define    MAX_PCR_PROPERTIES      (MAX_CAP_DATA/sizeof(TPMS_TAGGED_PCR_SELECT))
#define    MAX_ECC_CURVES          (MAX_CAP_DATA/sizeof(TPM_ECC_CURVE))
#endif
```

6.6 TPMB.h

This file contains extra TPM2B structures

```
#ifndef _TPMB_H
#define _TPMB_H
#include "TPM_Types.h"
```

This macro helps avoid having to type in the structure in order to create a new TPM2B type that is used in a function.

```
#define TPM2B_TYPE(name, bytes)           \
    typedef union {                       \
        struct  {                         \
            UINT16  size;                 \
            BYTE    buffer[(bytes)];       \
        } t;                              \
        TPM2B   b;                        \
    } TPM2B_##name
```

Macro to instance and initialize a TPM2B value

```
#define TPM2B_INIT(TYPE, name)  \
    TPM2B_##TYPE    name = {sizeof(name.t.buffer), {0}}
```

A 2B structure for a seed

```
TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
```

A 2B hash block

```
TPM2B_TYPE(HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
TPM2B_TYPE(RSA_PRIME, MAX_RSA_KEY_BYTES/2);
TPM2B_TYPE(1_BYTE_VALUE, 1);
TPM2B_TYPE(2_BYTE_VALUE, 2);
TPM2B_TYPE(4_BYTE_VALUE, 4);
TPM2B_TYPE(20_BYTE_VALUE, 20);
TPM2B_TYPE(32_BYTE_VALUE, 32);
TPM2B_TYPE(48_BYTE_VALUE, 48);
TPM2B_TYPE(64_BYTE_VALUE, 64);
TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
#endif
```

6.7 TpmError.h

```
#ifndef _TPM_ERROR_H
#define _TPM_ERROR_H
#include "TpmBuildSwitches.h"
#define     FATAL_ERROR_ALLOCATION                  (1)
#define     FATAL_ERROR_DIVIDE_ZERO                 (2)
#define     FATAL_ERROR_INTERNAL                    (3)
#define     FATAL_ERROR_PARAMETER                   (4)
#define     FATAL_ERROR_ENTROPY                     (5)
#define     FATAL_ERROR_SELF_TEST                   (6)
#define     FATAL_ERROR_CRYPTO                      (7)
#define     FATAL_ERROR_NV_UNRECOVERABLE            (8)
#define     FATAL_ERROR_REMANUFACTURED              (9) // indicates that the TPM has
                                                        // been re-manufactured after an
                                                        // unrecoverable NV error
#define     FATAL_ERROR_DRBG                        (10)
#define     FATAL_ERROR_FORCED                      (666)
```

These are the crypto assertion routines. When a function returns an unexpected and unrecoverable result, the assertion fails and the TpmFail() is called

```
void
TpmFail(const char *function, int line, int code);
typedef void    (*FAIL_FUNCTION)(const char *, int, int);
#define FAIL(a) (TpmFail(__FUNCTION__, __LINE__, a))
#if defined(EMPTY_ASSERT)
#   define pAssert(a)  ((void)0)
#else
#   define pAssert(a) (!!(a) ? 1 : (FAIL(FATAL_ERROR_PARAMETER), 0))
#endif
#endif // _TPM_ERROR_H
```

6.8 Global.h

6.8.1 Description

This file contains internal global type definitions and data declarations that are need between subsystems. The instantiation of global data is in Global.c. The initialization of global data is in the subsystem that is the primary owner of the data.

The first part of this file has the typedefs for structures and other defines used in many portions of the code. After the typedef clause, is a clause that defines global values that are only present in RAM. The next three clauses define the structures for the NV data areas: persistent, orderly, and state save. Additional clauses define the data that is used in specific modules. That data is private to the module but is collected here to simplify the management of the instance data. All the data is instanced in Global.c.

6.8.2 Includes

```
#ifndef         GLOBAL_H
#define         GLOBAL_H
//#define SELF_TEST
#include        "TpmBuildSwitches.h"
#include        "Tpm.h"
#include        "TPMB.h"
#include        "CryptoEngine.h"
#include        <setjmp.h>
```

6.8.3 Defines and Types

6.8.3.1 Unreferenced Parameter

This define is used to eliminate the compiler warning about an unreferenced parameter. Basically, it tells the compiler that it is not an accident that the parameter is unreferenced.

```
#ifndef UNREFERENCED_PARAMETER
#   define UNREFERENCED_PARAMETER(a)   (a)
#endif
#include    "bits.h"
```

6.8.3.2 Crypto Self-Test Values

Define these values here if the AlgorithmTests() project is not used

```
#ifndef SELF_TEST
extern ALGORITHM_VECTOR     g_implementedAlgorithms;
extern ALGORITHM_VECTOR     g_toTest;
#else
LIB_IMPORT extern ALGORITHM_VECTOR     g_implementedAlgorithms;
LIB_IMPORT extern ALGORITHM_VECTOR     g_toTest;
#endif
```

These macros are used in CryptUtil() to invoke the incremental self test.

```
#define     TEST(alg) if(TEST_BIT(alg, g_toTest)) CryptTestAlgorithm(alg, NULL)
```

Use of TPM_ALG_NULL is reserved for RSAEP/RSADP testing. If someone is wanting to test a hash with that value, don't do it.

```
#define     TEST_HASH(alg)                                              \
            if(     TEST_BIT(alg, g_toTest)                             \
                &&  (alg != ALG_NULL_VALUE))                            \
                CryptTestAlgorithm(alg, NULL)
```

6.8.3.3 Hash and HMAC State Structures

These definitions are for the types that can be in a hash state structure. These types are used in the crypto utilities

```
typedef BYTE    HASH_STATE_TYPE;
#define HASH_STATE_EMPTY        ((HASH_STATE_TYPE) 0)
```

```
#define HASH_STATE_HASH          ((HASH_STATE_TYPE) 1)
#define HASH_STATE_HMAC          ((HASH_STATE_TYPE) 2)
```

A HASH_STATE structure contains an opaque hash stack state. A caller would use this structure when performing incremental hash operations. The state is updated on each call. If *type* is an HMAC_STATE, or HMAC_STATE_SEQUENCE then state is followed by the HMAC key in *oPad* format.

```
typedef struct
{
    CPRI_HASH_STATE     state;              // hash state
    HASH_STATE_TYPE     type;               // type of the context
} HASH_STATE;
```

An HMAC_STATE structure contains an opaque HMAC stack state. A caller would use this structure when performing incremental HMAC operations. This structure contains a hash state and an HMAC key and allows slightly better stack optimization than adding an HMAC key to each hash state.

```
typedef struct
{
    HASH_STATE          hashState;          // the hash state
    TPM2B_HASH_BLOCK    hmacKey;            // the HMAC key
} HMAC_STATE;
```

6.8.3.4 Other Types

An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)

```
typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

```
typedef BYTE    TIME_INFO[sizeof(TPMS_TIME_INFO)];
```

A NAME is a BYTE array that can contain a TPMU_NAME

```
typedef BYTE    NAME[sizeof(TPMU_NAME)];
```

6.8.4 Loaded Object Structures

6.8.4.1 Description

The structures in clause 6.8.4 define the object layout as it exists in TPM memory.

Two types of objects are defined: an ordinary object such as a key, and a sequence object that may be a hash, HMAC, or event.

6.8.4.2 OBJECT_ATTRIBUTES

An OBJECT_ATTRIBUTES structure contains the variable attributes of an object. These properties are not part of the public properties but are used by the TPM in managing the object. An OBJECT_ATTRIBUTES is used in the definition of the OBJECT data type.

```
typedef struct
{
    unsigned            publicOnly   : 1;   //0) SET if only the public portion of
                                            //   an object is loaded
    unsigned            epsHierarchy : 1;   //1) SET if the object belongs to EPS
                                            //   Hierarchy
```



```
    unsigned             ppsHierarchy : 1;    //2) SET if the object belongs to PPS
                                              //   Hierarchy
    unsigned             spsHierarchy : 1;    //3) SET f the object belongs to SPS
                                              //   Hierarchy
    unsigned             evict        : 1;    //4) SET if the object is a platform or
                                              //   owner evict object.  Platform-
                                              //   evict object belongs to PPS
                                              //   hierarchy, owner-evict object
                                              //   belongs to SPS or EPS hierarchy.
                                              //   This bit is also used to mark a
                                              //   completed sequence object so it
                                              //   will be flush when the
                                              //   SequenceComplete command succeeds.
    unsigned             primary     : 1;     //5) SET for a primary object
    unsigned             temporary   : 1;     //6) SET for a temporary object
    unsigned             stClear     : 1;     //7) SET for an stClear object
    unsigned             hmacSeq     : 1;     //8) SET for an HMAC sequence object
    unsigned             hashSeq     : 1;     //9) SET for a hash sequence object
    unsigned             eventSeq    : 1;     //10) SET for an event sequence object
    unsigned             ticketSafe  : 1;     //11) SET if a ticket is safe to create
                                              //    for hash sequence object
    unsigned             firstBlock  : 1;     //12) SET if the first block of hash
                                              //    data has been received.  It
                                              //    works with ticketSafe bit
    unsigned             isParent    : 1;     //13) SET if the key has the proper
                                              //    attributes to be a parent key
    unsigned             privateExp  : 1;     //14) SET when the private exponent
                                              //    of an RSA key has been validated.
    unsigned         reserved    : 1;     //15) reserved bits. unused.
} OBJECT_ATTRIBUTES;
```

6.8.4.3 OBJECT Structure

An OBJECT structure holds the object public, sensitive, and meta-data associated. This structure is implementation dependent. For this implementation, the structure is not optimized for space but rather for clarity of the reference implementation. Other implementations may choose to overlap portions of the structure that are not used simultaneously. These changes would necessitate changes to the source code but those changes would be compatible with the reference implementation.

```
typedef struct
{
    // The attributes field is required to be first followed by the publicArea.
    // This allows the overlay of the object structure and a sequence structure
    OBJECT_ATTRIBUTES    attributes;          // object attributes
    TPMT_PUBLIC          publicArea;          // public area of an object
    TPMT_SENSITIVE       sensitive;           // sensitive area of an object

#ifdef  TPM_ALG_RSA
    TPM2B_PUBLIC_KEY_RSA privateExponent;     // Additional field for the private
                                              // exponent of an RSA key.
#endif
    TPM2B_NAME           qualifiedName;       // object qualified name
    TPMI_DH_OBJECT       evictHandle;         // if the object is an evict object,
                                              // the original handle is kept here.
                                              // The 'working' handle will be the
                                              // handle of an object slot.

    TPM2B_NAME           name;                // Name of the object name. Kept here
                                              // to avoid repeatedly computing it.
} OBJECT;
```

6.8.4.4 HASH_OBJECT Structure

This structure holds a hash sequence object or an event sequence object.

The first four components of this structure are manually set to be the same as the first four components of the object structure. This prevents the object from being inadvertently misused as sequence objects occupy the same memory as a regular object. A debug check is present to make sure that the offsets are what they are supposed to be.

```
typedef struct
{
    OBJECT_ATTRIBUTES   attributes;         // The attributes of the HASH object
    TPMI_ALG_PUBLIC     type;               // algorithm
    TPMI_ALG_HASH       nameAlg;            // name algorithm
    TPMA_OBJECT         objectAttributes;   // object attributes

    // The data below is unique to a sequence object
    TPM2B_AUTH          auth;               // auth for use of sequence
    union
    {
        HASH_STATE      hashState[HASH_COUNT];
        HMAC_STATE      hmacState;
    }                   state;
} HASH_OBJECT;
```

6.8.4.5 ANY_OBJECT

This is the union for holding either a sequence object or a regular object.

```
typedef union
{
    OBJECT              entity;
    HASH_OBJECT         hash;
} ANY_OBJECT;
```

6.8.5 AUTH_DUP Types

These values are used in the authorization processing.

```
typedef UINT32          AUTH_ROLE;
#define AUTH_NONE       ((AUTH_ROLE)(0))
#define AUTH_USER       ((AUTH_ROLE)(1))
#define AUTH_ADMIN      ((AUTH_ROLE)(2))
#define AUTH_DUP        ((AUTH_ROLE)(3))
```

6.8.6 Active Session Context

6.8.6.1 Description

The structures in clause 6.8.6 define the internal structure of a session context.

6.8.6.2 SESSION_ATTRIBUTES

The attributes in the SESSION_ATTRIBUTES structure track the various properties of the session. It maintains most of the tracking state information for the policy session. It is used within the SESSION structure.

```
typedef struct
```



```
{
    unsigned            isPolicy : 1;        //1) SET if the session may only
                                             //   be used for policy
    unsigned            isAudit : 1;         //2) SET if the session is used
                                             //   for audit
    unsigned            isBound : 1;         //3) SET if the session is bound to
                                             //   with an entity.
                                             //   This attribute will be CLEAR if
                                             //   either isPolicy or isAudit is SET.
    unsigned            iscpHashDefined : 1;//4) SET if the cpHash has been defined
                                             //   This attribute is not SET unless
                                             //   'isPolicy' is SET.
    unsigned            isAuthValueNeeded : 1;
                                             //5) SET if the authValue is required
                                             //   for computing the session HMAC.
                                             //   This attribute is not SET unless
                                             //   isPolicy is SET.
    unsigned            isPasswordNeeded : 1;
                                             //6) SET if a password authValue is
                                             //   required for authorization
                                             //   This attribute is not SET unless
                                             //   isPolicy is SET.
    unsigned            isPPRequired : 1;    //7) SET if physical presence is
                                             //   required to be asserted when the
                                             //   authorization is checked.
                                             //   This attribute is not SET unless
                                             //   isPolicy is SET.
    unsigned            isTrialPolicy : 1;   //8) SET if the policy session is
                                             //   created for trial of the policy's
                                             //   policyHash generation.
                                             //   This attribute is not SET unless
                                             //   isPolicy is SET.
    unsigned            isDaBound : 1;       //9) SET if the bind entity had noDA
                                             //   CLEAR. If this is SET, then an
                                             //   auth failure using this session
                                             //   will count against lockout even
                                             //   if the object being authorized is
                                             //   exempt from DA.
    unsigned            isLockoutBound : 1; //10)SET if the session is bound to
                                             //   lockoutAuth.
    unsigned            requestWasBound : 1;//11) SET if the session is being used
                                             //    with the bind entity. If SET
                                             //    the authValue will not be use
                                             //    in the response HMAC computation.
    unsigned            checkNvWritten : 1; //12) SET if the TPMA_NV_WRITTEN
                                             //    attribute needs to be checked
                                             //    when the policy is used for
                                             //    authorization for NV access.
                                             //    If this is SET for any other
                                             //    type, the policy will fail.
    unsigned            nvWrittenState : 1; //13) SET if TPMA_NV_WRITTEN is
                                             //    required to be SET.
} SESSION_ATTRIBUTES;
```


6.8.6.3 SESSION Structure

The SESSION structure contains all the context of a session except for the associated *contextID*.

NOTE The *contextID* of a session is only relevant when the session context is stored off the TPM.

```
typedef struct
{
    TPM_ALG_ID          authHashAlg;         // session hash algorithm
    TPM2B_NONCE         nonceTPM;            // last TPM-generated nonce for
```

```
                                                // this session

    TPMT_SYM_DEF        symmetric;              // session symmetric algorithm (if any)
    TPM2B_AUTH          sessionKey;             // session secret value used for
                                                // generating HMAC and encryption keys

    SESSION_ATTRIBUTES  attributes;             // session attributes
    TPM_CC              commandCode;            // command code (policy session)
    TPMA_LOCALITY       commandLocality;        // command locality (policy session)
    UINT32              pcrCounter;             // PCR counter value when PCR is
                                                // included (policy session)
                                                // If no PCR is included, this
                                                // value is 0.

    UINT64              startTime;              // value of TPMS_CLOCK_INFO.clock when
                                                // the session was started (policy
                                                // session)

    UINT64              timeOut;                // timeout relative to
                                                // TPMS_CLOCK_INFO.clock
                                                // There is no timeout if this value
                                                // is 0.
    union
    {
        TPM2B_NAME      boundEntity;             // value used to track the entity to
                                                 // which the session is bound

        TPM2B_DIGEST    cpHash;                  // the required cpHash value for the
                                                 // command being authorized

    } u1;                                        // 'boundEntity' and 'cpHash' may
                                                 // share the same space to save memory

    union
    {
        TPM2B_DIGEST    auditDigest;            // audit session digest
        TPM2B_DIGEST    policyDigest;             // policyHash

    } u2;                                       // audit log and policyHash may
                                                // share space to save memory
} SESSION;
```

6.8.7 PCR

6.8.7.1 PCR_SAVE Structure

The PCR_SAVE structure type contains the PCR data that are saved across power cycles. Only the static PCR are required to be saved across power cycles. The DRTM and resettable PCR are not saved. The number of static and resettable PCR is determined by the platform-specific specification to which the TPM is built.

```
typedef struct
{
#ifdef TPM_ALG_SHA1
    BYTE                sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA256
    BYTE                sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA384
    BYTE                sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA512
```



```
    BYTE                sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SM3_256
    BYTE                sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
#endif

    // This counter increments whenever the PCR are updated.
    // NOTE: A platform-specific specification may designate
    //       certain PCR changes as not causing this counter
    //       to increment.
    UINT32              pcrCounter;

} PCR_SAVE;
```

6.8.7.2 PCR_POLICY

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```
typedef struct
{
    TPMI_ALG_HASH       hashAlg[NUM_POLICY_PCR_GROUP];
    TPM2B_DIGEST        a;
    TPM2B_DIGEST        policy[NUM_POLICY_PCR_GROUP];
} PCR_POLICY;
```

6.8.7.3 PCR_AUTHVALUE

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```
typedef struct
{
    TPM2B_DIGEST        auth[NUM_AUTHVALUE_PCR_GROUP];
} PCR_AUTHVALUE;
```

6.8.8 Startup

6.8.8.1 SHUTDOWN_NONE

ISO/IEC 11889-2 defines the two shutdown/startup types that may be used in TPM2_Shutdown() and TPM2_Startup(). This additional define is used by the TPM to indicate that no shutdown was received.

NOTE This is a reserved value.

```
#define SHUTDOWN_NONE   (TPM_SU)(0xFFFF)
```

6.8.8.2 STARTUP_TYPE

This enumeration is the possible startup types. The type is determined by the combination of TPM2_ShutDown() and TPM2_Startup().

```
typedef enum
{
    SU_RESET,
    SU_RESTART,
    SU_RESUME
} STARTUP_TYPE;
```

6.8.9 NV

6.8.9.1 NV_RESERVE

This enumeration defines the master list of the elements of a reserved portion of NV. This list includes all the pre-defined data that takes space in NV, either as persistent data or as state save data. The enumerations are used as indexes into an array of offset values. The offset values then are used to index into NV. This is method provides an imperfect analog to an actual NV implementation.

```
typedef enum
{
// Entries below mirror the PERSISTENT_DATA structure. These values are written
// to NV as individual items.
    // hierarchy
    NV_DISABLE_CLEAR,
    NV_OWNER_ALG,
    NV_ENDORSEMENT_ALG,
    NV_LOCKOUT_ALG,
    NV_OWNER_POLICY,
    NV_ENDORSEMENT_POLICY,
    NV_LOCKOUT_POLICY,
    NV_OWNER_AUTH,
    NV_ENDORSEMENT_AUTH,
    NV_LOCKOUT_AUTH,

    NV_EP_SEED,
    NV_SP_SEED,
    NV_PP_SEED,

    NV_PH_PROOF,
    NV_SH_PROOF,
    NV_EH_PROOF,

    // Time
    NV_TOTAL_RESET_COUNT,
    NV_RESET_COUNT,

    // PCR
    NV_PCR_POLICIES,
    NV_PCR_ALLOCATED,

    // Physical Presence
    NV_PP_LIST,

    // Dictionary Attack
    NV_FAILED_TRIES,
    NV_MAX_TRIES,
    NV_RECOVERY_TIME,
    NV_LOCKOUT_RECOVERY,
    NV_LOCKOUT_AUTH_ENABLED,

    // Orderly State flag
    NV_ORDERLY,

    // Command Audit
    NV_AUDIT_COMMANDS,
    NV_AUDIT_HASH_ALG,
    NV_AUDIT_COUNTER,

    // Algorithm Set
    NV_ALGORITHM_SET,

    NV_FIRMWARE_V1,
    NV_FIRMWARE_V2,
```



```
// The entries above are in PERSISTENT_DATA. The entries below represent
// structures that are read and written as a unit.

// ORDERLY_DATA data structure written on each orderly shutdown
    NV_ORDERLY_DATA,

// STATE_CLEAR_DATA structure written on each Shutdown(STATE)
    NV_STATE_CLEAR,

// STATE_RESET_DATA structure written on each Shutdown(STATE)
    NV_STATE_RESET,

    NV_RESERVE_LAST             // end of NV reserved data list
} NV_RESERVE;
```

6.8.9.2 NV_INDEX

The NV_INDEX structure defines the internal format for an NV index. The *indexData* size varies according to the type of the index. In this implementation, all of the index is manipulated as a unit.

```
typedef struct
{
    TPMS_NV_PUBLIC      publicArea;
    TPM2B_AUTH          authValue;
} NV_INDEX;
```

6.8.10 COMMIT_INDEX_MASK

This is the define for the mask value that is used when manipulating the bits in the commit bit array. The commit counter is a 64-bit value and the low order bits are used to index the *commitArray*. This mask value is applied to the commit counter to extract the bit number in the array.

```
#ifdef TPM_ALG_ECC
#define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
#endif
```

6.8.11 RAM Global Values

6.8.11.1 Description

The values in clause 6.8.11 are only extant in RAM. They are defined here and instanced in Global.c.

6.8.11.2 g_rcIndex

This array is used to contain the array of values that are added to a return code when it is a parameter-, handle-, or session-related error. This is an implementation choice and the same result can be achieved by using a macro.

```
extern const UINT16     g_rcIndex[15];
```

6.8.11.3 g_exclusiveAuditSession

This location holds the session handle for the current exclusive audit session. If there is no exclusive audit session, the location is set to TPM_RH_UNASSIGNED.

```
extern TPM_HANDLE       g_exclusiveAuditSession;
```

6.8.11.4 g_time

This value is the count of milliseconds since the TPM was powered up. This value is initialized at _TPM_Init().

```
extern  UINT64          g_time;
```

6.8.11.5 g_phEnable

This is the platform hierarchy control and determines if the platform hierarchy is available. This value is SET on each TPM2_Startup(). The default value is SET.

```
extern BOOL             g_phEnable;
```

6.8.11.6 g_pceReConfig

This value is SET if a TPM2_PCR_Allocate() command successfully executed since the last TPM2_Startup(). If so, then the next shutdown is required to be Shutdown(CLEAR).

```
extern BOOL             g_pcrReConfig;
```

6.8.11.7 g_DRTMHandle

This location indicates the sequence object handle that holds the DRTM sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence DRTM sequence is started on either _TPM_Init() or _TPM_Hash_Start().

```
extern TPMI_DH_OBJECT   g_DRTMHandle;
```

6.8.11.8 g_DrtmPreStartup

This value indicates that an H-CRTM occurred after _TPM_Init() but before TPM2_Startup(). The define is used to add the *g_DrtmPreStartup* value to *gp_orderlyState* at shutdown. This is a bit of a hack that was done to avoid adding another NV variable just to have a bit

```
extern  BOOL            g_DrtmPreStartup;
#define PRE_STARTUP_FLAG    0x8000
```

6.8.11.9 g_updateNV

This flag indicates if NV should be updated at the end of a command. This flag is set to FALSE at the beginning of each command in ExecuteCommand(). This flag is checked in ExecuteCommand() after the detailed actions of a command complete. If the command execution was successful and this flag is SET, any pending NV writes will be committed to NV.

```
extern BOOL             g_updateNV;
```

6.8.11.10 g_clearOrderly

This flag indicates if the execution of a command should cause the orderly state to be cleared. This flag is set to FALSE at the beginning of each command in ExecuteCommand() and is checked in ExecuteCommand() after the detailed actions of a command complete but before the check of *g_updateNV*. If this flag is TRUE, and the orderly state is not SHUTDOWN_NONE, then the orderly state in NV memory will be changed to SHUTDOWN_NONE.


```
extern BOOL             g_clearOrderly;
```

6.8.11.11 g_prevOrderlyState

This location indicates how the TPM was shut down before the most recent TPM2_Startup(). This value, along with the startup type, determines if the TPM should do a TPM Reset, TPM Restart, or TPM Resume.

```
extern TPM_SU           g_prevOrderlyState;
```

6.8.11.12 g_nvOk

This value indicates if the NV integrity check was successful or not. If not and the failure was severe, then the TPM would have been put into failure mode after it had been re-manufactured. If the NV failure was in the area where the state-save data is kept, then this variable will have a value of FALSE indicating that a TPM2_Startup(CLEAR) is required.

```
extern BOOL             g_nvOk;
```

6.8.11.13 g_platformUnique

This location contains the unique value(s) used to identify the TPM. It is loaded on every _TPM2_Startup() The first value is used to seed the RNG. The second value is used as a vendor *authValue*. The value used by the RNG would be the value derived from the chip unique value (such as fused) with a dependency on the authorities of the code in the TPM boot path. The second would be derived from the chip unique value with a dependency on the details of the code in the boot path. That is, the first value depends on the various signers of the code and the second depends on what was signed. The TMP vendor should not be able to know the first value but they are expected to know the second.

```
extern TPM2B_AUTH       g_platformUniqueAuthorities; // Reserved for RNG
extern TPM2B_AUTH       g_platformUniqueDetails;   // referenced by VENDOR_PERMANENT
```

6.8.12 Persistent Global Values

6.8.12.1 Description

The values in clause 6.8.12 are global values that are persistent across power events. The lifetime of the values determines the structure in which the value is placed.

6.8.12.2 PERSISTENT_DATA

This structure holds the persistent values that only change as a consequence of a specific Protected Capability and are not affected by TPM power events (TPM2_Startup() or TPM2_Shutdown().

```
typedef struct
{
//*********************************************************************************
//          Hierarchy
//*********************************************************************************
// The values in this clause are related to the hierarchies.

    BOOL                disableClear;       // TRUE if TPM2_Clear() using
                                            // lockoutAuth is disabled

    // Hierarchy authPolicies
    TPMI_ALG_HASH       ownerAlg;
```

```
    TPMI_ALG_HASH       endorsementAlg;
    TPMI_ALG_HASH       lockoutAlg;
    TPM2B_DIGEST        ownerPolicy;
    TPM2B_DIGEST        endorsementPolicy;
    TPM2B_DIGEST        lockoutPolicy;

    // Hierarchy authValues
    TPM2B_AUTH          ownerAuth;
    TPM2B_AUTH          endorsementAuth;
    TPM2B_AUTH          lockoutAuth;

    // Primary Seeds
    TPM2B_SEED          EPSeed;
    TPM2B_SEED          SPSeed;
    TPM2B_SEED          PPSeed;
    // Note there is a nullSeed in the state_reset memory.

    // Hierarchy proofs
    TPM2B_AUTH          phProof;
    TPM2B_AUTH          shProof;
    TPM2B_AUTH          ehProof;
    // Note there is a nullProof in the state_reset memory.

//*****************************************************************************
//          Reset Events
//*****************************************************************************
// A count that increments at each TPM reset and never get reset during the life
// time of TPM.  The value of this counter is initialized to 1 during TPM
// manufacture process.
    UINT64              totalResetCount;

// This counter increments on each TPM Reset. The counter is reset by
// TPM2_Clear().
    UINT32              resetCount;


//*****************************************************************************
//          PCR
//*****************************************************************************
// This structure hold the policies for those PCR that have an update policy.
// This implementation only supports a single group of PCR controlled by
// policy. If more are required, then this structure would be changed to
// an array.
    PCR_POLICY          pcrPolicies;

// This structure indicates the allocation of PCR. The structure contains a
// list of PCR allocations for each implemented algorithm. If no PCR are
// allocated for an algorithm, a list entry still exists but the bit map
// will contain no SET bits.
    TPML_PCR_SELECTION  pcrAllocated;

//*****************************************************************************
//          Physical Presence
//*****************************************************************************
// The PP_LIST type contains a bit map of the commands that require physical
// to be asserted when the authorization is evaluated. Physical presence will be
// checked if the corresponding bit in the array is SET and if the authorization
// handle is TPM_RH_PLATFORM.
//
// These bits may be changed with TPM2_PP_Commands().
    BYTE                ppList[((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7)/8];

//*****************************************************************************
//          Dictionary attack values
//*****************************************************************************
// These values are used for dictionary attack tracking and control.
```



```
    UINT32              failedTries;        // the current count of unexpired
                                            // authorization failures

    UINT32              maxTries;           // number of unexpired authorization
                                            // failures before the TPM is in
                                            // lockout

    UINT32              recoveryTime;       // time between authorization failures
                                            // before failedTries is decremented

    UINT32              lockoutRecovery;    // time that must expire between
                                            // authorization failures associated
                                            // with lockoutAuth

    BOOL                lockOutAuthEnabled; // TRUE if use of lockoutAuth is
                                            // allowed

//*****************************************************************************
//              Orderly State
//*****************************************************************************
// The orderly state for current cycle
    TPM_SU              orderlyState;

//*****************************************************************************
//             Command audit values.
//*****************************************************************************
    BYTE                auditComands[((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8];
    TPMI_ALG_HASH       auditHashAlg;
    UINT64              auditCounter;

//*****************************************************************************
//             Algorithm selection
//*****************************************************************************
//
// The 'algorithmSet' value indicates the collection of algorithms that are
// currently in used on the TPM.  The interpretation of value is vendor dependent.
    UINT32              algorithmSet;

//*****************************************************************************
//             Firmware version
//*****************************************************************************
// The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is
// a scheme used in development to allow determination of the linker build time
// of the TPM. An actual implementation would implement these values in a way that
// is consistent with vendor needs. The values are maintained in RAM for simplified
// access with a master version in NV.  These values are modified in a
// vendor-specific way.

// g_firmwareV1 contains the more significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex
// value, it will have the format of yyyymmdd
    UINT32              firmwareV1;

// g_firmwareV1 contains the less significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex
// value, it will have the format of 00 hh mm ss
    UINT32              firmwareV2;

} PERSISTENT_DATA;
extern PERSISTENT_DATA  gp;
```


6.8.12.3 ORDERLY_DATA

The data in this structure is saved to NV on each TPM2_Shutdown().

```
typedef struct orderly_data
{

//*****************************************************************************
//           TIME
//*****************************************************************************

// Clock has two parts. One is the state save part and one is the NV part. The
// state save version is updated on each command. When the clock rolls over, the
// NV version is updated. When the TPM starts up, if the TPM was shutdown in and
// orderly way, then the sClock value is used to initialize the clock. If the
// TPM shutdown was not orderly, then the persistent value is used and the safe
// attribute is clear.

    UINT64              clock;              // The orderly version of clock
    TPMI_YES_NO         clockSafe;          // Indicates if the clock value is
                                            // safe.
//*********************************************************************************
//          DRBG
//*********************************************************************************
#ifdef _DRBG_STATE_SAVE
    // This is DRBG state data. This is saved each time the value of clock is
    // updated.
    DRBG_STATE          drbgState;
#endif

} ORDERLY_DATA;
extern ORDERLY_DATA     go;
```

6.8.12.4 STATE_CLEAR_DATA

This structure contains the data that is saved on Shutdown(STATE). and restored on Startup(STATE). The values are set to their default settings on any Startup(Clear). In other words the data is only persistent across TPM Resume.

If the comments associated with a parameter indicate a default reset value, the value is applied on each Startup(CLEAR).

```
typedef struct state_clear_data
{
//*****************************************************************************
//           Hierarchy Control
//*****************************************************************************
    BOOL                shEnable;           // default reset is SET
    BOOL                ehEnable;           // default reset is SET
    BOOL                phEnableNV;         // default reset is SET
    TPMI_ALG_HASH       platformAlg;        // default reset is TPM_ALG_NULL
    TPM2B_DIGEST        platformPolicy;     // default reset is an Empty Buffer
    TPM2B_AUTH          platformAuth;       // default reset is an Empty Buffer


//*****************************************************************************
//           PCR
//*****************************************************************************
// The set of PCR to be saved on Shutdown(STATE)
    PCR_SAVE            pcrSave;            // default reset is 0...0

// This structure hold the authorization values for those PCR that have an
// update authorization.
// This implementation only supports a single group of PCR controlled by
// authorization. If more are required, then this structure would be changed to
// an array.
    PCR_AUTHVALUE       pcrAuthValues;

```



```
} STATE_CLEAR_DATA;
extern STATE_CLEAR_DATA gc;
```

6.8.12.5 State Reset Data

This structure contains data is that is saved on Shutdown(STATE) and restored on the subsequent Startup(ANY). That is, the data is preserved across TPM Resume and TPM Restart.

If a default value is specified in the comments this value is applied on TPM Reset.

```
typedef struct state_reset_data
{
//*****************************************************************************
//           Hierarchy Control
//*****************************************************************************
    TPM2B_AUTH          nullProof;          // The proof value associated with
                                            // the TPM_RH_NULL hierarchy. The
                                            // default reset value is from the RNG.

    TPM2B_SEED          nullSeed;           // The seed value for the TPM_RN_NULL
                                            // hierarchy. The default reset value
                                            // is from the RNG.

//*****************************************************************************
//            Context
//*****************************************************************************
// The 'clearCount' counter is incremented each time the TPM successfully executes
// a TPM Resume. The counter is included in each saved context that has 'stClear'
// SET (including descendants of keys that have 'stClear' SET). This prevents these
// objects from being loaded after a TPM Resume.
// If 'clearCount' at its maximum value when the TPM receives a Shutdown(STATE),
// the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
    UINT32              clearCount;         // The default reset value is 0.

    UINT64              objectContextID;    // This is the context ID for a saved
                                            //  object context. The default reset
                                            //  value is 0.

    CONTEXT_SLOT        contextArray[MAX_ACTIVE_SESSIONS];
                                            // This is the value from which the
                                            // 'contextID' is derived. The
                                            // default reset value is {0}.


    CONTEXT_COUNTER     contextCounter;     // This array contains contains the
                                            // values used to track the version
                                            // numbers of saved contexts (see
                                            // Session.c in for details). The
                                            // default reset value is 0.

//*****************************************************************************
//            Command Audit
//*****************************************************************************
// When an audited command completes, ExecuteCommand() checks the return
// value.  If it is TPM_RC_SUCCESS, and the command is an audited command, the
// TPM will extend the cpHash and rpHash for the command to this value. If this
// digest was the Zero Digest before the cpHash was extended, the audit counter
// is incremented.

    TPM2B_DIGEST        commandAuditDigest; // This value is set to an Empty Digest
                                            // by TPM2_GetCommandAuditDigest() or a
                                            // TPM Reset.

//*****************************************************************************
//            Boot counter
```

```
//*****************************************************************************

    UINT32              restartCount;       // This counter counts TPM Restarts.
                                            // The default reset value is 0.

//*********************************************************************************
//            PCR
//*********************************************************************************
// This counter increments whenever the PCR are updated. This counter is preserved
// across TPM Resume even though the PCR are not preserved. This is because
// sessions remain active across TPM Restart and the count value in the session
// is compared to this counter so this counter must have values that are unique
// as long as the sessions are active.
// NOTE: A platform-specific specification may designate that certain PCR changes
//       do not increment this counter to increment.
    UINT32              pcrCounter;         // The default reset value is 0.

#ifdef TPM_ALG_ECC

//*****************************************************************************
//         ECDAA
//*****************************************************************************
    UINT64              commitCounter;      // This counter increments each time
                                            // TPM2_Commit() returns
                                            // TPM_RC_SUCCESS. The default reset
                                            // value is 0.


    TPM2B_NONCE         commitNonce;        // This random value is used to compute
                                            // the commit values. The default reset
                                            // value is from the RNG.

// This implementation relies on the number of bits in g_commitArray being a
// power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
    BYTE                 commitArray[16];   // The default reset value is {0}.

#endif //TPM_ALG_ECC

} STATE_RESET_DATA;
extern STATE_RESET_DATA gr;
```

6.8.13 Global Macro Definitions

This macro is used to ensure that a handle, session, or parameter number is only added if the response code is FMT1.

```
#define RcSafeAddToResult(r, v) \
    ((r) + (((r) & RC_FMT1) ? (v) : 0))
```

This macro is used when a parameter is not otherwise referenced in a function. This macro is normally not used by itself but is paired with a pAssert() within a #ifdef *pAssert*. If *pAssert* is not defined, then a parameter might not otherwise be referenced. This macro **uses** the parameter from the perspective of the compiler so it doesn't complain.

```
#define UNREFERENCED(a) ((void)(a))
```

6.9 Private data

```
#if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C
```

From SessionProcess.c

The following arrays are used to save command sessions information so that the command handle/session buffer does not have to be preserved for the duration of the command. These arrays are indexed by the session index in accordance with the order of sessions in the session area of the command.

Array of the authorization session handles

```
extern TPM_HANDLE       s_sessionHandles[MAX_SESSION_NUM];
```

Array of authorization session attributes

```
extern TPMA_SESSION     s_attributes[MAX_SESSION_NUM];
```

Array of handles authorized by the corresponding authorization sessions; and if none, then TPM_RH_UNASSIGNED value is used

```
extern TPM_HANDLE       s_associatedHandles[MAX_SESSION_NUM];
```

Array of nonces provided by the caller for the corresponding sessions

```
extern TPM2B_NONCE      s_nonceCaller[MAX_SESSION_NUM];
```

Array of authorization values (HMAC's or passwords) for the corresponding sessions

```
extern TPM2B_AUTH       s_inputAuthValues[MAX_SESSION_NUM];
```

Special value to indicate an undefined session index

```
#define             UNDEFINED_INDEX     (0xFFFF)
```

Index of the session used for encryption of a response parameter

```
extern UINT32           s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
extern UINT32           s_decryptSessionIndex;
```

Index of a session used for audit

```
extern UINT32           s_auditSessionIndex;
```

The *cpHash* for an audit session

```
extern TPM2B_DIGEST     s_cpHashForAudit;
```

The *cpHash* for command audit

```
#ifdef  TPM_CC_GetCommandAuditDigest
extern TPM2B_DIGEST    s_cpHashForCommandAudit;
#endif
```

Number of authorization sessions present in the command

```
extern UINT32           s_sessionNum;
```

Flag indicating if NV update is pending for the *lockOutAuthEnabled* or *failedTries* DA parameter

```
extern BOOL             s_DAPendingOnNV;
```

```
#endif // SESSION_PROCESS_C
#if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
```

From DA.c

This variable holds the accumulated time since the last time that *failedTries* was decremented. This value is in millisecond.

```
extern UINT64       s_selfHealTimer;
```

This variable holds the accumulated time that the *lockoutAuth* has been blocked.

```
extern UINT64       s_lockoutTimer;
#endif // DA_C
#if defined NV_C || defined GLOBAL_C
```

From NV.c

List of pre-defined address of reserved data

```
extern UINT32       s_reservedAddr[NV_RESERVE_LAST];
```

List of pre-defined reserved data size in byte

```
extern UINT32       s_reservedSize[NV_RESERVE_LAST];
```

Size of data in RAM index buffer

```
extern UINT32       s_ramIndexSize;
```

Reserved RAM space for frequently updated NV Index. The data layout in ram buffer is {NV_handle(), size of data, data} for each NV index data stored in RAM

```
extern BYTE      s_ramIndex[RAM_INDEX_SPACE];
```

Address of size of RAM index space in NV

```
extern UINT32   s_ramIndexSizeAddr;
```

Address of NV copy of RAM index space

```
extern UINT32   s_ramIndexAddr;
```

Address of maximum counter value; an auxiliary variable to implement NV counters

```
extern UINT32   s_maxCountAddr;
```

Beginning of NV dynamic area; starts right after the *s_maxCountAddr* and *s_evictHandleMapAddr* variables

```
extern UINT32   s_evictNvStart;
```

Beginning of NV dynamic area; also the beginning of the predefined reserved data area.

```
extern UINT32   s_evictNvEnd;
```

NV availability is sampled as the start of each command and stored here so that its value remains consistent during the command execution


```
extern TPM_RC   s_NvStatus;
#endif
#if defined OBJECT_C || defined GLOBAL_C
```

From Object.c

This type is the container for an object.

```
typedef struct
{
    BOOL          occupied;
    ANY_OBJECT        object;
} OBJECT_SLOT;
```

This is the memory that holds the loaded objects.

```
extern OBJECT_SLOT     s_objects[MAX_LOADED_OBJECTS];
#endif // OBJECT_C
#if defined PCR_C || defined GLOBAL_C
```

From PCR.c

```
typedef struct
{
#ifdef TPM_ALG_SHA1
    // SHA1 PCR
    BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA256
    // SHA256 PCR
    BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA384
    // SHA384 PCR
    BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SHA512
    // SHA512 PCR
    BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
#endif
#ifdef TPM_ALG_SM3_256
    // SHA256 PCR
    BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
#endif
} PCR;
typedef struct
{
    unsigned int    stateSave : 1;             // if the PCR value should be
                                               // saved in state save
    unsigned int    resetLocality : 5;         // The locality that the PCR
                                               // can be reset
    unsigned int    extendLocality : 5;        // The locality that the PCR
                                               // can be extend
} PCR_Attributes;
extern PCR          s_pcrs[IMPLEMENTATION_PCR];
#endif // PCR_C
#if defined SESSION_C || defined GLOBAL_C
```

From Session.c

Container for HMAC or policy session tracking information

```
typedef struct
{
```

```
    BOOL                occupied;
    SESSION             session;        // session structure
} SESSION_SLOT;
extern SESSION_SLOT     s_sessions[MAX_LOADED_SESSIONS];
```

The index in *conextArray* that has the value of the oldest saved session context. When no context is saved, this will have a value that is greater than or equal to MAX_ACTIVE_SESSIONS.

```
extern UINT32            s_oldestSavedSession;
```

The number of available session slot openings. When this is 1, a session can't be created or loaded if the GAP is maxed out. The exception is that the oldest saved session context can always be loaded (assuming that there is a space in memory to put it)

```
extern int               s_freeSessionSlots;
#endif // SESSION_C
```

From Manufacture.c

```
extern BOOL              g_manufactured;
#if defined POWER_C || defined GLOBAL_C
```

From Power.c

This value indicates if a TPM2_Startup() commands has been receive since the power on event. This flag is maintained in power simulation module because this is the only place that may reliably set this flag to FALSE.

```
extern BOOL              s_initialized;
#endif // POWER_C
#if defined MEMORY_LIB_C || defined GLOBAL_C
```

The *s_actionOutputBuffer* should not be modifiable by the host system until the TPM has returned a response code. The *s_actionOutputBuffer* should not be accessible until response parameter encryption, if any, is complete.

```
extern UINT32   s_actionInputBuffer[1024];          // action input buffer
extern UINT32   s_actionOutputBuffer[1024];         // action output buffer
extern BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
#endif // MEMORY_LIB_C
```

From TPMFail.c

This value holds the address of the string containing the name of the function in which the failure occurred. This address value isn't useful for anything other than helping the vendor to know in which file the failure occurred.

```
extern jmp_buf    g_jumpBuffer;           // the jump buffer
extern BOOL       g_inFailureMode;        // Indicates that the TPM is in failure mode
extern BOOL       g_forceFailureMode;     // flag to force failure mode during test
#if defined TPM_FAIL_C || defined GLOBAL_C || 1
extern UINT32     s_failFunction;
extern UINT32     s_failLine;             // the line in the file at which
                                          // the error was signaled
extern UINT32     s_failCode;             // the error code used
#endif // TPM_FAIL_C
#endif // GLOBAL_H
```


6.10 Tpm.h

Root header file for building any TPM. lib code

```
#ifndef     _TPM_H
#define     _TPM_H
#include    "bool.h"
#include    "Implementation.h"
#include    "TPM_Types.h"
#include    "swap.h"
#endif
```

6.11 swap.h

```
#ifndef _SWAP_H
#define _SWAP_H
#include "Implementation.h"
#if    NO_AUTO_ALIGN == YES || LITTLE_ENDIAN_TPM == YES
```

The aggregation macros for machines that do not allow unaligned access or for little-endian machines. Aggregate bytes into an UINT

```
#define BYTE_ARRAY_TO_UINT8(b)   (UINT8)((b)[0])
#define BYTE_ARRAY_TO_UINT16(b)  (UINT16)(  ((b)[0] <<  8) \
                                          +  (b)[1])
#define BYTE_ARRAY_TO_UINT32(b)  (UINT32)(  ((b)[0] << 24) \
                                          + ((b)[1] << 16) \
                                          + ((b)[2] << 8 ) \
                                          +  (b)[3])
#define BYTE_ARRAY_TO_UINT64(b)  (UINT64)(  ((UINT64)(b)[0] << 56) \
                                          + ((UINT64)(b)[1] << 48) \
                                          + ((UINT64)(b)[2] << 40) \
                                          + ((UINT64)(b)[3] << 32) \
                                          + ((UINT64)(b)[4] << 24) \
                                          + ((UINT64)(b)[5] << 16) \
                                          + ((UINT64)(b)[6] <<  8) \
                                          +  (UINT64)(b)[7])
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)      ((b)[0] = (BYTE)(i), i)
#define UINT16_TO_BYTE_ARRAY(i, b)     ((b)[0] = (BYTE)((i) >>  8), \
                                        (b)[1] = (BYTE) (i),        \
                                        (i))
#define UINT32_TO_BYTE_ARRAY(i, b)     ((b)[0] = (BYTE)((i) >> 24), \
                                        (b)[1] = (BYTE)((i) >> 16), \
                                        (b)[2] = (BYTE)((i) >>  8), \
                                        (b)[3] = (BYTE) (i),        \
                                        (i))
#define UINT64_TO_BYTE_ARRAY(i, b)     ((b)[0] = (BYTE)((i) >> 56), \
                                        (b)[1] = (BYTE)((i) >> 48), \
                                        (b)[2] = (BYTE)((i) >> 40), \
                                        (b)[3] = (BYTE)((i) >> 32), \
                                        (b)[4] = (BYTE)((i) >> 24), \
                                        (b)[5] = (BYTE)((i) >> 16), \
                                        (b)[6] = (BYTE)((i) >>  8), \
                                        (b)[7] = (BYTE) (i),        \
                                        (i))
#else
```

the big-endian macros for machines that allow unaligned memory access Aggregate a byte array into a UINT

```
#define BYTE_ARRAY_TO_UINT8(b)        *((UINT8  *)(b))
#define BYTE_ARRAY_TO_UINT16(b)       *((UINT16 *)(b))
#define BYTE_ARRAY_TO_UINT32(b)       *((UINT32 *)(b))
#define BYTE_ARRAY_TO_UINT64(b)       *((UINT64 *)(b))
```

Disaggregate a UINT into a byte array

```
#define UINT8_TO_BYTE_ARRAY(i, b)   (*((UINT8  *)(b)) = (i))
#define UINT16_TO_BYTE_ARRAY(i, b)  (*((UINT16 *)(b)) = (i))
#define UINT32_TO_BYTE_ARRAY(i, b)  (*((UINT32 *)(b)) = (i))
#define UINT64_TO_BYTE_ARRAY(i, b)  (*((UINT64 *)(b)) = (i))
#endif  // NO_AUTO_ALIGN == YES
#endif  // _SWAP_H
```

6.12 InternalRoutines.h

```
#ifndef     INTERNAL_ROUTINES_H
#define     INTERNAL_ROUTINES_H
```

NULL definition

```
#ifndef         NULL
#define         NULL        (0)
#endif
```

UNUSED_PARAMETER

```
#ifndef         UNUSED_PARAMETER
#define         UNUSED_PARAMETER(param)     (void)(param);
#endif
```

Internal data definition

```
#include "Global.h"
#include "VendorString.h"
```

Error Reporting

```
#include "TpmError.h"
```

DRTM functions

```
#include "_TPM_Hash_Start_fp.h"
#include "_TPM_Hash_Data_fp.h"
#include "_TPM_Hash_End_fp.h"
```

Internal subsystem functions

```
#include "Object_fp.h"
#include "Entity_fp.h"
#include "Session_fp.h"
#include "Hierarchy_fp.h"
#include "NV_fp.h"
#include "PCR_fp.h"
#include "DA_fp.h"
#include "TpmFail_fp.h"
```

Internal support functions

```
#include "CommandCodeAttributes_fp.h"
#include "MemoryLib_fp.h"
```



```
#include "marshal_fp.h"
#include "Time_fp.h"
#include "Locality_fp.h"
#include "PP_fp.h"
#include "CommandAudit_fp.h"
#include "Manufacture_fp.h"
#include "Power_fp.h"
#include "Handle_fp.h"
#include "Commands_fp.h"
#include "AlgorithmCap_fp.h"
#include "PropertyCap_fp.h"
#include "Bits_fp.h"
```

Internal crypto functions

```
#include "Ticket_fp.h"
#include "CryptUtil_fp.h"
#include "CryptSelfTest_fp.h"
#endif
```

6.13 TpmBuildSwitches.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

```
#ifndef _TPM_BUILD_SWITCHES_H
#define _TPM_BUILD_SWITCHES_H
#define SIMULATION
#define FIPS_COMPLIANT
```

Define the alignment macro appropriate for the build environment For MS C compiler

```
#define ALIGN_TO(boundary)  __declspec(align(boundary))
```

For ISO 9899:2011

```
// #define ALIGN_TO(boundary)   _Alignas(boundary)
```

This switch enables the RNG state save and restore

```
#undef  _DRBG_STATE_SAVE
#define _DRBG_STATE_SAVE        // Comment this out if no state save is wanted
```

Set the alignment size for the crypto. It would be nice to set this according to macros automatically defined by the build environment, but that doesn't seem possible because there isn't any simple set for that. So, this is just a plugged value. Your compiler should complain if this alignment isn't possible.

NOTE This value can be set at the command line or just plugged in here.

```
#ifdef CRYPTO_ALIGN_16
#   define CRYPTO_ALIGNMENT    16
#elif defined CRYPTO_ALIGN_8
#   define CRYPTO_ALIGNMENT    8
#eliF defined CRYPTO_ALIGN_2
#   define  CRYPTO_ALIGNMENT    2
#elif defined CRTYPO_ALIGN_1
#   define  CRYPTO_ALIGNMENT    1
#else
#   define CRYPTO_ALIGNMENT    4    // For 32-bit builds
#endif
#define CRYPTO_ALIGNED  ALIGN_TO(CRYPTO_ALIGNMENT)
```

This macro is used to handle LIB_EXPORT of function and variable names in lieu of a . def file

```
#define LIB_EXPORT __declspec(dllexport)
// #define LIB_EXPORT
```

For import of a variable

```
#define LIB_IMPORT __declspec(dllimport)
//#define LIB_IMPORT
```

This is defined to indicate a function that does not return. This is used in static code anlaysis.

```
#define __declspec(noreturn)
//#define #ifdef SELF_TEST
#pragma comment(lib, "algorithmtests.lib")
#endif
```

The switches in this group can only be enabled when running a simulation

```
#ifdef SIMULATION
#   define RSA_KEY_CACHE
#   define TPM_RNG_FOR_DEBUG
#else
#   undef RSA_KEY_CACHE
#   undef TPM_RNG_FOR_DEBUG
#endif  // SIMULATION
#define INLINE   __inline
#endif // _TPM_BUILD_SWITCHES_H
```

6.14 VendorString.h

```
#ifndef      _VENDOR_STRING_H
#define      _VENDOR_STRING_H
```

Define up to 4-byte values for MANUFACTURER. This value defines the response for TPM_PT_MANUFACTURER in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here.

```
#define     MANUFACTURER    "MSFT"
```

The following #if macro may be deleted after a proper MANUFACTURER is provided.

```
#ifndef MANUFACTURER
#error MANUFACTURER is not provided. \
Please modify include\VendorString.h to provide a specific \
manufacturer name.
#endif
```

Define up to 4, 4-byte values. The values must each be 4 bytes long and the last value used may contain trailing zeros. These values define the response for TPM_PT_VENDOR_STRING_(1-4) in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here. The vendor strings 2-4 may also be defined as appropriately.

```
#define        VENDOR_STRING_1       "xCG "
#define        VENDOR_STRING_2       "fTPM"
// #define        VENDOR_STRING_3
// #define        VENDOR_STRING_4
```

The following #if macro may be deleted after a proper VENDOR_STRING_1 is provided.


```
#ifndef VENDOR_STRING_1
#error VENDOR_STRING_1 is not provided. \
Please modify include\VendorString.h to provide a vednor specific \
string.
#endif
```

the more significant 32-bits of a vendor-specific value indicating the version of the firmware The following line should be un-commented and a vendor specific firmware V1 should be provided here. The FIRMWARE_V2 may also be defined as appropriate.

```
#define   FIRMWARE_V1         (0x20130315)
```

the less significant 32-bits of a vendor-specific value indicating the version of the firmware

```
#define   FIRMWARE_V2         (0x00120000)
```

The following #if macro may be deleted after a proper FIRMWARE_V1 is provided.

```
#ifndef FIRMWARE_V1
#error  FIRMWARE_V1 is not provided. \
Please modify include\VendorString.h to provide a vendor specific firmware \
version
#endif
#endif
```


7 Main

7.1 CommandDispatcher()

In the reference implementation, a program that uses ISO/IEC 11889-3 as input automatically generates the command dispatch code. The function prototype header file (CommandDispatcher_fp.h) is shown here.

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;
- invokes the function that performs the command actions;
- marshals the returned handles, if any; and
- marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

```
 #ifndef    COMMANDDISPATCHER_FP_H
#define    COMMANDDISPATCHER_FP_H
TPM_RC
CommandDispatcher(
    TPMI_ST_COMMAND_TAG     tag,     // IN: Input command tag
    TPM_CC      command_code,        // IN: Command code
    INT32       *parm_buffer_size,   // IN: size of parameter buffer
    BYTE        *parm_buffer_start,  // IN: pointer to start of parameter buffer
    TPM_HANDLE  handles[],           // IN: handle array
    UINT32      *res_handle_size,    // OUT: size of handle buffer in response
    UINT32      *res_parm_size       // OUT: size of parameter buffer in response
    );
#endif
```

7.2 ExecCommand.c

7.2.1 Introduction

This file contains the entry function ExecuteCommand() which provides the main control flow for TPM command execution.

7.2.2 Includes

```
#include "InternalRoutines.h"
#include "HandleProcess_fp.h"
#include "SessionProcess_fp.h"
#include "CommandDispatcher_fp.h"
```

Uncomment this next #include if doing static command/response buffer sizing

```
// #include "CommandResponseSizes_fp.h"
```

7.2.3 ExecuteCommand()

The function performs the following steps.

a) Parses the command header from input buffer.

b) Calls ParseHandleBuffer() to parse the handle area of the command.

c) Validates that each of the handles references a loaded entity.

d) Calls ParseSessionBuffer() () to:
   1) unmarshal and parse the session area;
   2) check the authorizations; and
   3) when necessary, decrypt a parameter.
e) Calls CommandDispatcher() to:
   1) unmarshal the command parameters from the command buffer;
   2) call the routine that performs the command actions; and
   3) marshal the responses into the response buffer.
f) If any error occurs in any of the steps above create the error response and return.
g) Calls BuildResponseSession() to:
   1) when necessary, encrypt a parameter
   2) build the response authorization sessions
   3) update the audit sessions and nonces
h) Assembles handle, parameter and session buffers for response and return.

```
LIB_EXPORT void
ExecuteCommand(
    unsigned int     requestSize,    // IN: command buffer size
    unsigned char   *request,        // IN: command buffer
    unsigned int    *responseSize,   // OUT: response buffer size
    unsigned char   **response       // OUT: response buffer
    )
{
    // Command local variables
    TPM_ST                  tag;            // these first three variables are the
    UINT32                  commandSize;
    TPM_CC                  commandCode = 0;

    BYTE                   *parmBufferStart;    // pointer to the first byte of an
                                                // optional parameter buffer

    UINT32                  parmBufferSize = 0;// number of bytes in parameter area

    UINT32                  handleNum = 0;      // number of handles unmarshaled into
                                                // the handles array

    TPM_HANDLE              handles[MAX_HANDLE_NUM];// array to hold handles in the
                                                // command. Only handles in the handle
                                                // area are stored here, not handles
                                                // passed as parameters.

    // Response local variables
    TPM_RC                  result;             // return code for the command

    TPM_ST                  resTag;             // tag for the response

    UINT32                  resHandleSize = 0; // size of the handle area in the
                                                // response. This is needed so that the
                                                // handle area can be skipped when
                                                // generating the rpHash.

    UINT32                  resParmSize = 0;    // the size of the response parameters
                                                // These values go in the rpHash.

    UINT32                  resAuthSize = 0;    // size of authorization area in the
```

```
                                                   // response

    INT32                   size;                  // remaining data to be unmarshaled
                                                   // or remaining space in the marshaling
                                                   // buffer

    BYTE                   *buffer;                // pointer into the buffer being used
                                                   // for marshaling or unmarshaling

    UINT32                  i;                      // local temp

// This next function call is used in development to size the command and response
// buffers. The values printed are the sizes of the internal structures and
// not the sizes of the canonical forms of the command response structures. Also,
// the sizes do not include the tag, commandCode, requestSize, or the authorization
// fields.
//CommandResponseSizes();

    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write. Note, that this needs to be done
    // even when in failure mode. Otherwise, g_updateNV would stay SET while in
    // Failure mode and the NB would be written on each call.
    g_updateNV = FALSE;
    g_clearOrderly = FALSE;


    // As of Sept 25, 2013, the failure mode handling has been incorporated in the
    // reference code. This implementation requires that the system support
    // setjmp/longjmp. This code is put here because of the complexity being
    // added to the platform and simulator code to deal with all the variations
    // of errors.
    if(g_inFailureMode)
    {
        // Do failure mode processing
        TpmFailureMode (requestSize, request, responseSize, response);
        return;
    }
    if(setjmp(g_jumpBuffer) != 0)
    {
        // Get here if we got a longjump putting us into failure mode
        g_inFailureMode = TRUE;
        result = TPM_RC_FAILURE;
        goto Fail;
    }

    // Assume that everything is going to work.
    result = TPM_RC_SUCCESS;


    // Query platform to get the NV state.  The result state is saved internally
    // and will be reported by NvIsAvailable(). The reference code requires that
    // accessibility of NV does not change during the execution of a command.
    // Specifically, if NV is available when the command execution starts and then
    // is not available later when it is necessary to write to NV, then the TPM
    // will go into failure mode.
    NvCheckState();

    // Due to the limitations of the simulation, TPM clock must be explicitly
    // synchronized with the system clock whenever a command is received.
    // This function call is not necessary in a hardware TPM. However, taking
    // a snapshot of the hardware timer at the beginning of the command allows
    // the time value to be consistent for the duration of the command execution.
    TimeUpdateToCurrent();

    // Any command through this function will unceremoniously end the
    // _TPM_Hash_Data/_TPM_Hash_End sequence.
```



```
    if(g_DRTMHandle != TPM_RH_UNASSIGNED)
        ObjectTerminateEvent();

    // Get command buffer size and command buffer.
    size = requestSize;
    buffer = request;

    // Parse command header: tag, commandSize and commandCode.
    // First parse the tag. The unmarshaling routine will validate
    // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
    result = TPMI_ST_COMMAND_TAG_Unmarshal(&tag, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // Unmarshal the commandSize indicator.
    result = UINT32_Unmarshal(&commandSize, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // On a TPM that receives bytes on a port, the number of bytes that were
    // received on that port is requestSize it must be identical to commandSize.
    // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
    // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
    // as the input processing (the function that receives the command bytes and
    // places them in the input buffer) would likely have the input truncated when
    // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
    if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
    {
        result = TPM_RC_COMMAND_SIZE;
        goto Cleanup;
    }

    // Unmarshal the command code.
    result = TPM_CC_Unmarshal(&commandCode, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // Check to see if the command is implemented.
    if(!CommandIsImplemented(commandCode))
    {
        result = TPM_RC_COMMAND_CODE;
        goto Cleanup;
    }

#if  FIELD_UPGRADE_IMPLEMENTED  == YES
    // If the TPM is in FUM, then the only allowed command is
    // TPM_CC_FieldUpgradeData.
    if(IsFieldUgradeMode() && (commandCode != TPM_CC_FieldUpgradeData))
    {
        result = TPM_RC_UPGRADE;
        goto Cleanup;
    }
    else
#endif
        // Excepting FUM, the TPM only accepts TPM2_Startup() after
        // _TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
        // is no longer allowed.
        if((   !TPMIsStarted() && commandCode != TPM_CC_Startup)
            || (TPMIsStarted() && commandCode == TPM_CC_Startup))
        {
            result = TPM_RC_INITIALIZE;
            goto Cleanup;
        }

    // Start regular command process.
    // Parse Handle buffer.
```

```
    result = ParseHandleBuffer(commandCode, &buffer, &size, handles, &handleNum);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // Number of handles retrieved from handle area should be less than
    // MAX_HANDLE_NUM.
    pAssert(handleNum <= MAX_HANDLE_NUM);

    // All handles in the handle area are required to reference TPM-resident
    // entities.
    for(i = 0; i < handleNum; i++)
    {
        result = EntityGetLoadStatus(&handles[i], commandCode);
        if(result != TPM_RC_SUCCESS)
        {
            if(result == TPM_RC_REFERENCE_H0)
                result = result + i;
            else
                result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
            goto Cleanup;
        }
    }

    // Authorization session handling for the command.
    if(tag == TPM_ST_SESSIONS)
    {
        BYTE        *sessionBufferStart;// address of the session area first byte
                                        // in the input buffer

        UINT32      authorizationSize;  // number of bytes in the session area

        // Find out session buffer size.
        result = UINT32_Unmarshal(&authorizationSize, &buffer, &size);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;

        // Perform sanity check on the unmarshaled value. If it is smaller than
        // the smallest possible session or larger than the remaining size of
        // the command, then it is an error. NOTE: This check could pass but the
        // session size could still be wrong. That will be determined after the
        // sessions are unmarshaled.
        if(   authorizationSize < 9
           || authorizationSize > (UINT32) size)
        {
            result = TPM_RC_SIZE;
            goto Cleanup;
        }

        // The sessions, if any, follows authorizationSize.
        sessionBufferStart = buffer;

        // The parameters follow the session area.
        parmBufferStart = sessionBufferStart + authorizationSize;

        // Any data left over after removing the authorization sessions is
        // parameter data. If the command does not have parameters, then an
        // error will be returned if the remaining size is not zero. This is
        // checked later.
        parmBufferSize = size - authorizationSize;

        // The actions of ParseSessionBuffer() are described in the introduction.
        result = ParseSessionBuffer(commandCode,
                                    handleNum,
                                    handles,
                                    sessionBufferStart,
                                    authorizationSize,
```

```
                                    parmBufferStart,
                                    parmBufferSize);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;
    }
    else
    {
        // Whatever remains in the input buffer is used for the parameters of the
        // command.
        parmBufferStart = buffer;
        parmBufferSize = size;

        // The command has no authorization sessions.
        // If the command requires authorizations, then CheckAuthNoSession() will
        // return an error.
        result = CheckAuthNoSession(commandCode, handleNum, handles,
                                    parmBufferStart, parmBufferSize);
        if(result != TPM_RC_SUCCESS)
            goto Cleanup;
    }

    // CommandDispatcher returns a response handle buffer and a response parameter
    // buffer if it succeeds. It will also set the parameterSize field in the
    // buffer if the tag is TPM_RC_SESSIONS.
    result = CommandDispatcher(tag,
                               commandCode,
                               (INT32 *) &parmBufferSize,
                               parmBufferStart,
                               handles,
                               &resHandleSize,
                               &resParmSize);
    if(result != TPM_RC_SUCCESS)
        goto Cleanup;

    // Build the session area at the end of the parameter area.
    BuildResponseSession(tag,
                         commandCode,
                         resHandleSize,
                         resParmSize,
                         &resAuthSize);

Cleanup:
    // This implementation loads an "evict" object to a transient object slot in
    // RAM whenever an "evict" object handle is used in a command so that the
    // access to any object is the same. These temporary objects need to be
    // cleared from RAM whether the command succeeds or fails.
    ObjectCleanupEvict();

Fail:
    // The response will contain at least a response header.
    *responseSize = sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC);

    // If the command completed successfully, then build the rest of the response.
    if(result == TPM_RC_SUCCESS)
    {
        // Outgoing tag will be the same as the incoming tag.
        resTag = tag;
        // The overall response will include the handles, parameters,
        // and authorizations.
        *responseSize += resHandleSize + resParmSize + resAuthSize;

        // Adding parameter size field.
        if(tag == TPM_ST_SESSIONS)
            *responseSize += sizeof(UINT32);

        if(   g_clearOrderly == TRUE
```

```
            && gp.orderlyState != SHUTDOWN_NONE)
        {
            gp.orderlyState = SHUTDOWN_NONE;
            NvWriteReserved(NV_ORDERLY, &gp.orderlyState);
            g_updateNV = TRUE;
        }
    }
    else
    {
        // The command failed.
        // If this was a failure due to a bad command tag, then need to return
        // an ISO/IEC 11889 (first edition) compatible response
        if(result == TPM_RC_BAD_TAG)
            resTag = TPM_ST_RSP_COMMAND;
        else
            // return ISO/IEC 11889 compatible response
            resTag = TPM_ST_NO_SESSIONS;
    }
    // Try to commit all the writes to NV if any NV write happened during this
    // command execution. This check should be made for both succeeded and failed
    // commands, because a failed one may trigger a NV write in DA logic as well.
    // This is the only place in the command execution path that may call the NV
    // commit. If the NV commit fails, the TPM should be put in failure mode.
    if(g_updateNV && !g_inFailureMode)
    {
        g_updateNV = FALSE;
        if(!NvCommit())
            FAIL(FATAL_ERROR_INTERNAL);
    }

    // Marshal the response header.
    buffer = MemoryGetResponseBuffer(commandCode);
    TPM_ST_Marshal(&resTag, &buffer, NULL);
    UINT32_Marshal((UINT32 *)responseSize, &buffer, NULL);
    pAssert(*responseSize <= MAX_RESPONSE_SIZE);
    TPM_RC_Marshal(&result, &buffer, NULL);

    *response = MemoryGetResponseBuffer(commandCode);

    // Clear unused bit in response buffer.
    MemorySet(*response + *responseSize, 0, MAX_RESPONSE_SIZE - *responseSize);

    return;
}
```

7.3 ParseHandleBuffer()

In the reference implementation, the routine for unmarshaling the command handles is automatically generated from ISO/IEC 11889-3 command tables. The prototype header file (HandleProcess_fp.h) is shown here.

```
#ifndef     HANDLEPROCESS_FP_H
#define     HANDLEPROCESS_FP_H
TPM_RC
ParseHandleBuffer(
    TPM_CC      command_code,
    BYTE        **handle_buffer_start,
    INT32       *buffer_remain_size,
    TPM_HANDLE  handles[],
    UINT32      *handle_num
    );
#endif
```


7.4 SessionProcess.c

7.4.1 Introduction

This file contains the subsystem that process the authorization sessions including implementation of the Dictionary Attack logic. ExecCommand() uses ParseSessionBuffer() to process the authorization session area of a command and BuildResponseSession() to create the authorization session area of a response.

7.4.2 Includes and Data Definitions

```
#define SESSION_PROCESS_C
#include "InternalRoutines.h"
#include "SessionProcess_fp.h"
#include "Platform.h"
```

7.4.3 Authorization Support Functions

7.4.3.1 IsDAExempted()

This function indicates if a handle is exempted from DA logic. A handle is exempted if it is

a) a primary seed handle,

b) an object with *noDA* bit SET,

c) an NV Index with TPMA_NV_NO_DA bit SET, or

d) a PCR handle.

Table 1

Return Value	Meaning
TRUE	handle is exempted from DA logic
FALSE	handle is not exempted from DA logic

```
BOOL
IsDAExempted(
    TPM_HANDLE        handle         // IN: entity handle
    )
{
    BOOL          result = FALSE;

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
            // DA protection.
            result =  (handle != TPM_RH_LOCKOUT);
            break;

        // When this function is called, a persistent object will have been loaded
        // into an object slot and assigned a transient handle.
        case TPM_HT_TRANSIENT:
        {
            OBJECT      *object;
            object = ObjectGet(handle);
            result = (object->publicArea.objectAttributes.noDA == SET);
            break;
        }
```

```
        case TPM_HT_NV_INDEX:
        {
            NV_INDEX        nvIndex;
            NvGetIndexInfo(handle, &nvIndex);
            result = (nvIndex.publicArea.attributes.TPMA_NV_NO_DA == SET);
            break;
        }
        case TPM_HT_PCR:
            // PCRs are always exempted from DA.
            result = TRUE;
            break;
        default:
            break;
    }
    return result;
}
```

7.4.3.2 IncrementLockout()

This function is called after an authorization failure that involves use of an *authValue*. If the entity referenced by the handle is not exempt from DA protection, then the *failedTries* counter will be incremented.

Table 2

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization failure that caused DA lockout to increment
TPM_RC_BAD_AUTH	authorization failure did not cause DA lockout to increment

```
static TPM_RC
IncrementLockout(
    UINT32           sessionIndex
    )
{
    TPM_HANDLE       handle = s_associatedHandles[sessionIndex];
    TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
    TPM_RC           result;
    SESSION         *session = NULL;


    // Don't increment lockout unless the handle associated with the session
    // is DA protected or the session is bound to a DA protected entity.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsDAExempted(handle))
            return TPM_RC_BAD_AUTH;

    }
    else
    {
        session = SessionGet(sessionHandle);
        // If the session is bound to lockout, then use that as the relevant
        // handle. This means that an auth failure with a bound session
        // bound to lockoutAuth will take precedence over any other
        // lockout check
        if(session->attributes.isLockoutBound == SET)
            handle = TPM_RH_LOCKOUT;

        if(   session->attributes.isDaBound == CLEAR
           && IsDAExempted(handle)
          )
```



```
           // If the handle was changed to TPM_RH_LOCKOUT, this will not return
           // TPM_RC_BAD_AUTH
            return TPM_RC_BAD_AUTH;

    }

    if(handle == TPM_RH_LOCKOUT)
    {
        pAssert(gp.lockOutAuthEnabled);
        gp.lockOutAuthEnabled = FALSE;
        // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
        // the lockout auth will be reset at startup.
        if(gp.lockoutRecovery != 0)
        {
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
            {
                // No NV access for now. Put the TPM in pending mode.
                s_DAPendingOnNV = TRUE;
            }
            else
            {
                // Update NV.
                NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
                g_updateNV = TRUE;
            }
        }
    }
    else
    {
        if(gp.recoveryTime != 0)
        {
            gp.failedTries++;
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
            {
                // No NV access for now.  Put the TPM in pending mode.
                s_DAPendingOnNV = TRUE;
            }
            else
            {
                // Record changes to NV.
                NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
                g_updateNV = TRUE;
            }
        }
    }

    // Register a DA failure and reset the timers.
    DARegisterFailure(handle);

    return TPM_RC_AUTH_FAIL;
}
```


7.4.3.3 IsSessionBindEntity()

This function indicates if the entity associated with the handle is the entity, to which this session is bound. The binding would occur by making the **bind** parameter in TPM2_StartAuthSession() not equal to TPM_RH_NULL. The binding only occurs if the session is an HMAC session. The bind value is a combination of the Name and the *authValue* of the entity.

Table 3

Return Value	Meaning
TRUE	handle points to the session start entity
FALSE	handle does not point to the session start entity

```
static BOOL
IsSessionBindEntity(
    TPM_HANDLE        associatedHandle,  // IN: handle to be authorized
    SESSION          *session            // IN: associated session
    )
{
    TPM2B_NAME     entity;              // The bind value for the entity

    // If the session is not bound, return FALSE.
    if(!session->attributes.isBound)
        return FALSE;

    // Compute the bind value for the entity.
    SessionComputeBoundEntity(associatedHandle, &entity);

    // Compare to the bind value in the session.
    session->attributes.requestWasBound =
            Memory2BEqual(&entity.b, &session->u1.boundEntity.b);
    return session->attributes.requestWasBound;
}
```

7.4.3.4 IsPolicySessionRequired()

Checks if a policy session is required for a command. If a command requires DUP or ADMIN role authorization, then the handle that requires that role is the first handle in the command. This simplifies this checking. If a new command is created that requires multiple ADMIN role authorizations, then it will have to be special-cased in this function. A policy session is required if:

a) the command requires the DUP role,

b) the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET, or

c) the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index.

d) The authorized entity is a PCR belonging to a policy group, and has its policy initialized.

Table 4

Return Value	Meaning
TRUE	policy session is required
FALSE	policy session is not required

```
static BOOL
IsPolicySessionRequired(
    TPM_CC            commandCode,    // IN: command code
    UINT32            sessionIndex    // IN: session index
    )
{
    AUTH_ROLE       role = CommandAuthRole(commandCode, sessionIndex);
    TPM_HT          type = HandleGetType(s_associatedHandles[sessionIndex]);

    if(role == AUTH_DUP)
```



```
            return TRUE;

        if(role == AUTH_ADMIN)
        {
            if(type == TPM_HT_TRANSIENT)
            {
                OBJECT      *object = ObjectGet(s_associatedHandles[sessionIndex]);

                if(object->publicArea.objectAttributes.adminWithPolicy == CLEAR)
                    return FALSE;
            }
            return TRUE;
        }

        if(type == TPM_HT_PCR)
        {
            if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
            {
                TPM2B_DIGEST        policy;
                TPMI_ALG_HASH       policyAlg;
                policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
                                             &policy);
                if(policyAlg != TPM_ALG_NULL)
                    return TRUE;
            }
        }
        return FALSE;
    }
```

7.4.3.5 IsAuthValueAvailable()

This function indicates if *authValue* is available and allowed for USER role authorization of an entity.

This function is similar to IsAuthPolicyAvailable() except that it does not check the size of the *authValue* as IsAuthPolicyAvailable() does (a null *authValue* is a valid auth, but a null policy is not a valid policy).

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Table 5

Return Value	Meaning
TRUE	*authValue* is available
FALSE	*authValue* is not available

```
    static BOOL
    IsAuthValueAvailable(
        TPM_HANDLE        handle,         // IN: handle of entity
        TPM_CC            commandCode,    // IN: commandCode
        UINT32            sessionIndex    // IN: session index
        )
    {
        BOOL              result = FALSE;
        // If a policy session is required, the entity can not be authorized by
        // authValue. However, at this point, the policy session requirement should
        // already have been checked.
        pAssert(!IsPolicySessionRequired(commandCode, sessionIndex));

        switch(HandleGetType(handle))
        {
            case TPM_HT_PERMANENT:
                switch(handle)
```

```
            {
                    // At this point hierarchy availability has already been
                    // checked so primary seed handles are always available here
                case TPM_RH_OWNER:
                case TPM_RH_ENDORSEMENT:
                case TPM_RH_PLATFORM:
#ifdef VENDOR_PERMANENT
                    // This vendor defined handle associated with the
                    // manufacturer's shared secret
                case VENDOR_PERMANENT:
#endif
                    // NullAuth is always available.
                case TPM_RH_NULL:
                     // At the point when authValue availability is checked, control
                    // path has already passed the DA check so LockOut auth is
                    // always available here
                case TPM_RH_LOCKOUT:

                    result = TRUE;
                    break;
                default:
                    // Otherwise authValue is not available.
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // A persistent object has already been loaded and the internal
            // handle changed.
            {
                OBJECT          *object;
                object = ObjectGet(handle);

                // authValue is always available for a sequence object.
                if(ObjectIsSequence(object))
                {
                    result =  TRUE;
                    break;
                }
                // authValue is available for an object if it has its sensitive
                // portion loaded and
                //  1. userWithAuth bit is SET, or
                //  2. ADMIN role is required
                if(   object->attributes.publicOnly == CLEAR
                   &&    (object->publicArea.objectAttributes.userWithAuth == SET
                      || (CommandAuthRole(commandCode, sessionIndex) == AUTH_ADMIN
                         &&  object->publicArea.objectAttributes.adminWithPolicy
                            == CLEAR)))
                    result = TRUE;
            }
            break;
        case TPM_HT_NV_INDEX:
            // NV Index.
            {
                NV_INDEX       nvIndex;
                NvGetIndexInfo(handle, &nvIndex);
                if(IsWriteOperation(commandCode))
                {
                    if (nvIndex.publicArea.attributes.TPMA_NV_AUTHWRITE == SET)
                        result = TRUE;

                }
                else
                {
                    if (nvIndex.publicArea.attributes.TPMA_NV_AUTHREAD == SET)
                        result = TRUE;
                }
```



```
            }
            break;
        case TPM_HT_PCR:
            // PCR handle.
            // authValue is always allowed for PCR
            result =  TRUE;
            break;
        default:
            // Otherwise, authValue is not available
            break;
    }
    return result;
}
```

7.4.3.6 IsAuthPolicyAvailable()

This function indicates if an *authPolicy* is available and allowed.

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Table 6

Return Value	Meaning
TRUE	*authPolicy* is available
FALSE	*authPolicy* is not available

```
static BOOL
IsAuthPolicyAvailable(
    TPM_HANDLE       handle,         // IN: handle of entity
    TPM_CC           commandCode,    // IN: commandCode
    UINT32           sessionIndex    // IN: session index
    )
{
    BOOL             result = FALSE;
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            switch(handle)
            {
                // At this point hierarchy availability has already been checked.
                case TPM_RH_OWNER:
                    if (gp.ownerPolicy.t.size != 0)
                        result = TRUE;
                    break;

                case TPM_RH_ENDORSEMENT:
                    if (gp.endorsementPolicy.t.size != 0)
                        result = TRUE;
                    break;

                case TPM_RH_PLATFORM:
                    if (gc.platformPolicy.t.size != 0)
                        result = TRUE;
                    break;
                default:
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            {
                // Object handle.
```

```
                // An evict object would already have been loaded and given a
                // transient object handle by this point.
                OBJECT  *object = ObjectGet(handle);
                // Policy authorization is not available for an object with only
                // public portion loaded.
                if(object->attributes.publicOnly == CLEAR)
                {
                    // Policy authorization is always available for an object but
                    // is never available for a sequence.
                    if(!ObjectIsSequence(object))
                        result = TRUE;
                }
                break;
            }
        case TPM_HT_NV_INDEX:
            // An NV Index.
            {
                NV_INDEX         nvIndex;
                NvGetIndexInfo(handle, &nvIndex);
                // If the policy size is not zero, check if policy can be used.
                if(nvIndex.publicArea.authPolicy.t.size != 0)
                {
                    // If policy session is required for this handle, always
                    // uses policy regardless of the attributes bit setting
                    if(IsPolicySessionRequired(commandCode, sessionIndex))
                        result = TRUE;
                    // Otherwise, the presence of the policy depends on the NV
                    // attributes.
                    else if(IsWriteOperation(commandCode))
                    {
                        if (   nvIndex.publicArea.attributes.TPMA_NV_POLICYWRITE
                            == SET)
                            result = TRUE;
                    }
                    else
                    {
                        if (    nvIndex.publicArea.attributes.TPMA_NV_POLICYREAD
                            ==  SET)
                            result = TRUE;
                    }
                }
            }
            break;
        case TPM_HT_PCR:
            // PCR handle.
            if(PCRPolicyIsAvailable(handle))
                result = TRUE;
            break;
        default:
            break;
    }
    return result;
}
```

7.4.4 Session Parsing Functions

7.4.4.1 ComputeCpHash()

This function computes the *cpHash* as defined in ISO/IEC 11889-2 and specified in ISO/IEC 11889-1.

```
static void
ComputeCpHash(
    TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
```



```
    TPM_CC           commandCode,       // IN: command code
    UINT32           handleNum,         // IN: number of handles
    TPM_HANDLE       handles[],         // IN: array of handles
    UINT32           parmBufferSize,    // IN: size of input parameter area
    BYTE            *parmBuffer,        // IN: input parameter area
    TPM2B_DIGEST    *cpHash,            // OUT: cpHash
    TPM2B_DIGEST    *nameHash           // OUT: name hash of command
    )
{
    UINT32           i;
    HASH_STATE       hashState;
    TPM2B_NAME       name;

    // cpHash = hash(commandCode [ || authName1
    //                           [ || authName2
    //                           [ || authName 3 ]]]
    //                           [ || parameters])
    // A cpHash can contain just a commandCode only if the lone session is
    // an audit session.

    // Start cpHash.
    cpHash->t.size = CryptStartHash(hashAlg, &hashState);

    //  Add commandCode.
    CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);

    //  Add authNames for each of the handles.
    for(i = 0; i < handleNum; i++)
    {
        name.t.size = EntityGetName(handles[i], &name.t.name);
        CryptUpdateDigest2B(&hashState, &name.b);
    }

    //  Add the parameters.
    CryptUpdateDigest(&hashState, parmBufferSize, parmBuffer);

    //  Complete the hash.
    CryptCompleteHash2B(&hashState, &cpHash->b);

    // If the nameHash is needed, compute it here.
    if(nameHash != NULL)
    {
        // Start name hash. hashState may be reused.
        nameHash->t.size = CryptStartHash(hashAlg, &hashState);

        //  Adding names.
        for(i = 0; i < handleNum; i++)
        {
            name.t.size = EntityGetName(handles[i], &name.t.name);
            CryptUpdateDigest2B(&hashState, &name.b);
        }
        //  Complete hash.
        CryptCompleteHash2B(&hashState, &nameHash->b);
    }
    return;
}
```

7.4.4.2 CheckPWAuthSession()

This function validates the authorization provided in a PWAP session. It compares the input value to *authValue* of the authorized entity. Argument *sessionIndex* is used to get handles handle of the referenced entities from *s_inputAuthValues*[] and *s_associatedHandles*[].

Table 7

Error Returns	Meaning
TPM_RC_AUTH_FAIL	auth fails and increments DA failure count
TPM_RC_BAD_AUTH	auth fails but DA does not apply

```
static TPM_RC
CheckPWAuthSession(
    UINT32            sessionIndex    // IN: index of session to be processed
    )
{
    TPM2B_AUTH      authValue;
    TPM_HANDLE      associatedHandle = s_associatedHandles[sessionIndex];

    // Strip trailing zeros from the password.
    MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);

    // Get the auth value and size.
    authValue.t.size = EntityGetAuthValue(associatedHandle, &authValue.t.buffer);

    // Success if the digests are identical.
    if(Memory2BEqual(&s_inputAuthValues[sessionIndex].b, &authValue.b))
    {
        return TPM_RC_SUCCESS;
    }
    else                        // if the digests are not identical
    {
        // Invoke DA protection if applicable.
        return IncrementLockout(sessionIndex);
    }
}
```

7.4.4.3 ComputeCommandHMAC()

This function computes the HMAC for an authorization session in a command.

```
static void
ComputeCommandHMAC(
    UINT32            sessionIndex,   // IN: index of session to be processed
    TPM2B_DIGEST     *cpHash,         // IN: cpHash
    TPM2B_DIGEST     *hmac            // OUT: authorization HMAC
    )
{
    TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
    TPM2B_KEY         key;
    BYTE              marshalBuffer[sizeof(TPMA_SESSION)];
    BYTE             *buffer;
    UINT32            marshalSize;
    HMAC_STATE        hmacState;
    TPM2B_NONCE      *nonceDecrypt;
    TPM2B_NONCE      *nonceEncrypt;
    SESSION          *session;
    TPM_HT            sessionHandleType =
                          HandleGetType(s_sessionHandles[sessionIndex]);

    nonceDecrypt = NULL;
    nonceEncrypt = NULL;

    // Determine if extra nonceTPM values are going to be required.
    // If this is the first session (sessionIndex = 0) and it is an authorization
    // session that uses an HMAC, then check if additional session nonces are to be
    // included.
```



```
    if(   sessionIndex == 0
       && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
    {
        // If there is a decrypt session and if this is not the decrypt session,
        // then an extra nonce may be needed.
        if(   s_decryptSessionIndex != UNDEFINED_INDEX
           && s_decryptSessionIndex != sessionIndex)
        {
            // Will add the nonce for the decrypt session.
            SESSION *decryptSession
                        = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
            nonceDecrypt = &decryptSession->nonceTPM;
        }
        // Now repeat for the encrypt session.
        if(   s_encryptSessionIndex != UNDEFINED_INDEX
           && s_encryptSessionIndex != sessionIndex
           && s_encryptSessionIndex != s_decryptSessionIndex)
        {
            // Have to have the nonce for the encrypt session.
            SESSION *encryptSession
                        = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
            nonceEncrypt = &encryptSession->nonceTPM;
        }
    }

    // Continue with the HMAC processing.
    session = SessionGet(s_sessionHandles[sessionIndex]);

    // Generate HMAC key.
    MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));

    // Check if the session has an associated handle and if the associated entity
    // is the one to which the session is bound. If not, add the authValue of
    // this entity to the HMAC key.
    // If the session is bound to the object or the session is a policy session
    // with no authValue required, do not include the authValue in the HMAC key.
    // Note: For a policy session, its isBound attribute is CLEARED.

    // If the session isn't used for authorization, then there is no auth value
    // to add
    if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
    {
        // used for auth so see if this is a policy session with authValue needed
        // or an hmac session that is not bound
        if(         sessionHandleType == TPM_HT_POLICY_SESSION
                &&  session->attributes.isAuthValueNeeded == SET
            ||      sessionHandleType == TPM_HT_HMAC_SESSION
                && !IsSessionBindEntity(s_associatedHandles[sessionIndex], session)
          )
        {
            // add the authValue to the HMAC key
            pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>sizeof(key.t.buffer));
            key.t.size =   key.t.size
                         + EntityGetAuthValue(s_associatedHandles[sessionIndex],
                                         (AUTH_VALUE *)&(key.t.buffer[key.t.size]));
        }
    }

     // if the HMAC key size is 0, a NULL string HMAC is allowed
     if(    key.t.size == 0
         && s_inputAuthValues[sessionIndex].t.size == 0)
     {
         hmac->t.size = 0;
         return;
     }

```

```
    // Start HMAC
    hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);

    //  Add cpHash
    CryptUpdateDigest2B(&hmacState, &cpHash->b);

    //  Add nonceCaller
    CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);

    //  Add nonceTPM
    CryptUpdateDigest2B(&hmacState, &session->nonceTPM.b);

    //  If needed, add nonceTPM for decrypt session
    if(nonceDecrypt != NULL)
        CryptUpdateDigest2B(&hmacState, &nonceDecrypt->b);

    //  If needed, add nonceTPM for encrypt session
    if(nonceEncrypt != NULL)
        CryptUpdateDigest2B(&hmacState, &nonceEncrypt->b);

    //  Add sessionAttributes
    buffer = marshalBuffer;
    marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
                                       &buffer, NULL);
    CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);

    // Complete the HMAC computation
    CryptCompleteHMAC2B(&hmacState, &hmac->b);

    return;
}
```

7.4.4.4 CheckSessionHMAC()

This function checks the HMAC of in a session. It uses ComputeCommandHMAC() to compute the expected HMAC value and then compares the result with the HMAC in the authorization session. The authorization is successful if they are the same.

If the authorizations are not the same, IncrementLockout() is called. It will return TPM_RC_AUTH_FAIL if the failure caused the *failureCount* to increment. Otherwise, it will return TPM_RC_BAD_AUTH.

Table 8

Error Returns	Meaning
TPM_RC_AUTH_FAIL	auth failure caused *failureCount* increment
TPM_RC_BAD_AUTH	auth failure did not cause *failureCount* increment

```
static TPM_RC
CheckSessionHMAC(
    UINT32           sessionIndex,   // IN: index of session to be processed
    TPM2B_DIGEST    *cpHash          // IN: cpHash of the command
    )
{
    TPM2B_DIGEST         hmac;           // authHMAC for comparing

    // Compute authHMAC
    ComputeCommandHMAC(sessionIndex, cpHash, &hmac);

    // Compare the input HMAC with the authHMAC computed above.
    if(!Memory2BEqual(&s_inputAuthValues[sessionIndex].b,  &hmac.b))
    {
        // If an HMAC session has a failure, invoke the anti-hammering
```



```
            // if it applies to the authorized entity or the session.
            // Otherwise, just indicate that the authorization is bad.
            return IncrementLockout(sessionIndex);
        }
        return TPM_RC_SUCCESS;
    }
```

7.4.4.5 CheckPolicyAuthSession()

This function is used to validate the authorization in a policy session. This function performs the following comparisons to see if a policy authorization is properly provided. The check are:

a) compare *policyDigest* in session with *authPolicy* associated with the entity to be authorized;

b) compare timeout if applicable;

c) compare *commandCode* if applicable;

d) compare *cpHash* if applicable; and

e) see if PCR values have changed since computed.

If all the above checks succeed, the handle is authorized. The order of these comparisons is not important because any failure will result in the same error code.

Table 9

Error Returns	Meaning
TPM_RC_PCR_CHANGED	PCR value is not current
TPM_RC_POLICY_FAIL	policy session fails
TPM_RC_LOCALITY	command locality is not allowed
TPM_RC_POLICY_CC	CC doesn't match
TPM_RC_EXPIRED	policy session has expired
TPM_RC_PP	PP is required but not asserted
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```
static TPM_RC
CheckPolicyAuthSession(
    UINT32            sessionIndex,   // IN: index of session to be processed
    TPM_CC            commandCode,    // IN: command code
    TPM2B_DIGEST     *cpHash,         // IN: cpHash using the algorithm of this
                                      //     session
    TPM2B_DIGEST     *nameHash        // IN: nameHash using the session algorithm
    )
{
    TPM_RC            result = TPM_RC_SUCCESS;
    SESSION          *session;
    TPM2B_DIGEST      authPolicy;
    TPMI_ALG_HASH     policyAlg;
    UINT8             locality;

    // Initialize pointer to the auth session.
    session = SessionGet(s_sessionHandles[sessionIndex]);

    // If the command is TPM_RC_PolicySecret(), make sure that
    // either password or authValue is required
    if(     commandCode == TPM_CC_PolicySecret
        &&  session->attributes.isPasswordNeeded == CLEAR
```

```
        &&  session->attributes.isAuthValueNeeded == CLEAR)
        return TPM_RC_MODE;

    // See if the PCR counter for the session is still valid.
    if( !SessionPCRValueIsCurrent(s_sessionHandles[sessionIndex]) )
        return TPM_RC_PCR_CHANGED;

    // Get authPolicy.
    policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
                                    &authPolicy);
    // Compare authPolicy.
    if(!Memory2BEqual(&session->u2.policyDigest.b, &authPolicy.b))
        return TPM_RC_POLICY_FAIL;

    // Policy is OK so check if the other factors are correct

    // Compare policy hash algorithm.
    if(policyAlg != session->authHashAlg)
        return TPM_RC_POLICY_FAIL;

    // Compare timeout.
    if(session->timeOut != 0)
    {
        // Cannot compare time if clock stop advancing.  An TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;

        if(session->timeOut < go.clock)
            return TPM_RC_EXPIRED;
    }

    // If command code is provided it must match
    if(session->commandCode != 0)
    {
        if(session->commandCode != commandCode)
            return TPM_RC_POLICY_CC;
    }
    else
    {
        // If command requires a DUP or ADMIN authorization, the session must have
        // command code set.
        AUTH_ROLE   role = CommandAuthRole(commandCode, sessionIndex);
        if(role == AUTH_ADMIN || role == AUTH_DUP)
            return TPM_RC_POLICY_FAIL;
    }
    // Check command locality.
    {
        BYTE         sessionLocality[sizeof(TPMA_LOCALITY)];
        BYTE        *buffer = sessionLocality;

        // Get existing locality setting in canonical form
        TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);

        // See if the locality has been set
        if(sessionLocality[0] != 0)
        {
            // If so, get the current locality
            locality = _plat__LocalityGet();
            if (locality < 5)
            {
                if(    ((sessionLocality[0] & (1 << locality)) == 0)
                    || sessionLocality[0] > 31)
                    return TPM_RC_LOCALITY;
            }
```

```
            else if (locality > 31)
            {
                if(sessionLocality[0] != locality)
                    return TPM_RC_LOCALITY;
            }
            else
            {
                // Could throw an assert here but a locality error is just
                // as good. It just means that, whatever the locality is, it isn't
                // the locality requested so...
                return TPM_RC_LOCALITY;
            }
        }
    } // end of locality check

    // Check physical presence.
    if(   session->attributes.isPPRequired == SET
       && !_plat__PhysicalPresenceAsserted())
        return TPM_RC_PP;

    // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
    // DUP role for this handle.
    if(session->u1.cpHash.b.size != 0)
    {
        if(session->attributes.iscpHashDefined)
        {
            // Compare cpHash.
            if(!Memory2BEqual(&session->u1.cpHash.b, &cpHash->b))
                return TPM_RC_POLICY_FAIL;
        }
        else
        {
            // Compare nameHash.
            // When cpHash is not defined, nameHash is placed in its space.
            if(!Memory2BEqual(&session->u1.cpHash.b, &nameHash->b))
                return TPM_RC_POLICY_FAIL;
        }
    }
    if(session->attributes.checkNvWritten)
    {
        NV_INDEX        nvIndex;

        // If this is not an NV index, the policy makes no sense so fail it.
        if(HandleGetType(s_associatedHandles[sessionIndex])!= TPM_HT_NV_INDEX)
            return TPM_RC_POLICY_FAIL;

        // Get the index data
        NvGetIndexInfo(s_associatedHandles[sessionIndex], &nvIndex);

        // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
        if(    (nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
           != (session->attributes.nvWrittenState == SET))
           return TPM_RC_POLICY_FAIL;
    }

    return TPM_RC_SUCCESS;
}
```


7.4.4.6 RetrieveSessionData()

This function will unmarshal the sessions in the session area of a command. The values are placed in the arrays that are defined at the beginning of this file. The normal unmarshaling errors are possible.

Table 10

Error Returns	Meaning
TPM_RC_SUCCSS	unmarshaled without error
TPM_RC_SIZE	the number of bytes unmarshaled is not the same as the value for *authorizationSize* in the command

```
static TPM_RC
RetrieveSessionData (
    TPM_CC           commandCode,    // IN: command code
    UINT32          *sessionCount,   // OUT: number of sessions found
    BYTE            *sessionBuffer,  // IN: pointer to the session buffer
    INT32            bufferSize      // IN: size of the session buffer
    )
{
    int             sessionIndex;
    int             i;
    TPM_RC          result;
    SESSION        *session;
    TPM_HT          sessionType;

    s_decryptSessionIndex = UNDEFINED_INDEX;
    s_encryptSessionIndex = UNDEFINED_INDEX;
    s_auditSessionIndex = UNDEFINED_INDEX;

    for(sessionIndex = 0; bufferSize > 0; sessionIndex++)
    {
        // If maximum allowed number of sessions has been parsed, return a size
        // error with a session number that is larger than the number of allowed
        // sessions
        if(sessionIndex == MAX_SESSION_NUM)
            return TPM_RC_SIZE + TPM_RC_S + g_rcIndex[sessionIndex+1];

        // make sure that the associated handle for each session starts out
        // unassigned
        s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;

        // First parameter: Session handle.
        result = TPMI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
                                                &sessionBuffer, &bufferSize, TRUE);
        if(result != TPM_RC_SUCCESS)
            return result + TPM_RC_S + g_rcIndex[sessionIndex];

        // Second parameter: Nonce.
        result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
                                       &sessionBuffer, &bufferSize);
        if(result != TPM_RC_SUCCESS)
            return result + TPM_RC_S + g_rcIndex[sessionIndex];

        // Third parameter: sessionAttributes.
        result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
                                        &sessionBuffer, &bufferSize);
        if(result != TPM_RC_SUCCESS)
            return result + TPM_RC_S + g_rcIndex[sessionIndex];

        // Fourth parameter: authValue (PW or HMAC).
        result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
                                      &sessionBuffer, &bufferSize);
        if(result != TPM_RC_SUCCESS)
            return result + TPM_RC_S + g_rcIndex[sessionIndex];

        if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
        {
            // A PWAP session needs additional processing.
```

```
            //      Can't have any attributes set other than continueSession bit
            if(    s_attributes[sessionIndex].encrypt
                || s_attributes[sessionIndex].decrypt
                || s_attributes[sessionIndex].audit
                || s_attributes[sessionIndex].auditExclusive
                || s_attributes[sessionIndex].auditReset
              )
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            //      The nonce size must be zero.
            if(s_nonceCaller[sessionIndex].t.size != 0)
                return TPM_RC_NONCE + TPM_RC_S + g_rcIndex[sessionIndex];

            continue;
        }
        // For not password sessions...

        // Find out if the session is loaded.
        if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
            return TPM_RC_REFERENCE_S0 + sessionIndex;

        sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
        session = SessionGet(s_sessionHandles[sessionIndex]);
        // Check if the session is an HMAC/policy session.
        if(    (    session->attributes.isPolicy == SET
                 && sessionType == TPM_HT_HMAC_SESSION
               )
            || (    session->attributes.isPolicy == CLEAR
                 &&  sessionType == TPM_HT_POLICY_SESSION
               )
          )
            return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];

        // Check that this handle has not previously been used.
        for(i = 0; i < sessionIndex; i++)
        {
            if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
                return TPM_RC_HANDLE + TPM_RC_S + g_rcIndex[sessionIndex];
        }

        // If the session is used for parameter encryption or audit as well, set
        // the corresponding indices.

        // First process decrypt.
        if(s_attributes[sessionIndex].decrypt)
        {
            // Check if the commandCode allows command parameter encryption.
            if(DecryptSize(commandCode) == 0)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // Encrypt attribute can only appear in one session
            if(s_decryptSessionIndex != UNDEFINED_INDEX)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
            if(session->symmetric.algorithm == TPM_ALG_NULL)
                return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];

            // All checks passed, so set the index for the session used to decrypt
            // a command parameter.
            s_decryptSessionIndex = sessionIndex;
        }

        // Now process encrypt.
        if(s_attributes[sessionIndex].encrypt)
        {
```

```
            // Check if the commandCode allows response parameter encryption.
            if(EncryptSize(commandCode) == 0)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // Encrypt attribute can only appear in one session.
            if(s_encryptSessionIndex != UNDEFINED_INDEX)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
            if(session->symmetric.algorithm == TPM_ALG_NULL)
                return TPM_RC_SYMMETRIC + TPM_RC_S + g_rcIndex[sessionIndex];

            // All checks passed, so set the index for the session used to encrypt
            // a response parameter.
            s_encryptSessionIndex = sessionIndex;
        }

        // At last process audit.
        if(s_attributes[sessionIndex].audit)
        {
            // Audit attribute can only appear in one session.
            if(s_auditSessionIndex != UNDEFINED_INDEX)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // An audit session can not be policy session.
            if(   HandleGetType(s_sessionHandles[sessionIndex])
               == TPM_HT_POLICY_SESSION)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // If this is a reset of the audit session, or the first use
            // of the session as an audit session, it doesn't matter what
            // the exclusive state is. The session will become exclusive.
            if(   s_attributes[sessionIndex].auditReset == CLEAR
               && session->attributes.isAudit == SET)
            {
                // Not first use or reset. If auditExlusive is SET, then this
                // session must be the current exclusive session.
                if(   s_attributes[sessionIndex].auditExclusive == SET
                   && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
                   return TPM_RC_EXCLUSIVE;
            }

            s_auditSessionIndex = sessionIndex;
        }

        // Initialize associated handle as undefined. This will be changed when
        // the handles are processed.
        s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;

    }


    // Set the number of sessions found.
    *sessionCount = sessionIndex;
    return TPM_RC_SUCCESS;
}
```

7.4.4.7 CheckLockedOut()

This function checks to see if the TPM is in lockout. This function should only be called if the entity being checked is subject to DA protection. The TPM is in lockout if the NV is not available and a DA write is pending. Otherwise the TPM is locked out if checking for *lockoutAuth (lockoutAuthCheck* == TRUE) and use of *lockoutAuth* is disabled, or *failedTries* >= *maxTries.*

Table 11

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting
TPM_RC_NV_UNAVAILABLE	NV is not available at this time
TPM_RC_LOCKOUT	TPM is in lockout

```
static TPM_RC
CheckLockedOut(
    BOOL                lockoutAuthCheck   // IN: TRUE if checking is for lockoutAuth
    )
{
    TPM_RC      result;

    // If NV is unavailable, and current cycle state recorded in NV is not
    // SHUTDOWN_NONE, refuse to check any authorization because we would
    // not be able to handle a DA failure.
    result = NvIsAvailable();
    if(result != TPM_RC_SUCCESS && gp.orderlyState != SHUTDOWN_NONE)
        return result;

    // Check if DA info needs to be updated in NV.
    if(s_DAPendingOnNV)
    {
        // If NV is accessible, ...
        if(result == TPM_RC_SUCCESS)
        {
            // ... write the pending DA data and proceed.
            NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED,
                            &gp.lockOutAuthEnabled);
            NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
            g_updateNV = TRUE;
            s_DAPendingOnNV = FALSE;
        }
        else
        {
            // Otherwise no authorization can be checked.
            return result;
        }
    }

    // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
    // is disabled...
    if(lockoutAuthCheck)
    {
        if(gp.lockOutAuthEnabled == FALSE)
            return TPM_RC_LOCKOUT;
    }
    else
    {
        // ... or if the number of failed tries has been maxed out.
        if(gp.failedTries >= gp.maxTries)
            return TPM_RC_LOCKOUT;
    }
    return TPM_RC_SUCCESS;
}
```

7.4.4.8 CheckAuthSession()

This function checks that the authorization session properly authorizes the use of the associated handle.

Table 12

Error Returns	Meaning
TPM_RC_LOCKOUT	entity is protected by DA and TPM is in lockout, or TPM is locked out on NV update pending on DA parameters
TPM_RC_PP	Physical Presence is required but not provided
TPM_RC_AUTH_FAIL	HMAC or PW authorization failed with DA side-effects (can be a policy session)
TPM_RC_BAD_AUTH	HMAC or PW authorization failed without DA side-effects (can be a policy session)
TPM_RC_POLICY_FAIL	if policy session fails
TPM_RC_POLICY_CC	command code of policy was wrong
TPM_RC_EXPIRED	the policy session has expired
TPM_RC_PCR	???
TPM_RC_AUTH_UNAVAILABLE	*authValue* or *authPolicy* unavailable

```
static TPM_RC
CheckAuthSession(
    TPM_CC           commandCode,    // IN: commandCode
    UINT32           sessionIndex,   // IN: index of session to be processed
    TPM2B_DIGEST    *cpHash,         // IN: cpHash
    TPM2B_DIGEST    *nameHash        // IN: nameHash
    )
{
    TPM_RC           result;
    SESSION         *session = NULL;
    TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
    TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
    TPM_HT           sessionHandleType = HandleGetType(sessionHandle);

    pAssert(sessionHandle != TPM_RH_UNASSIGNED);

    if(sessionHandle != TPM_RS_PW)
        session = SessionGet(sessionHandle);

    pAssert(sessionHandleType != TPM_HT_POLICY_SESSION || session != NULL);

    // If the authorization session is not a policy session, or if the policy
    // session requires authorization, then check lockout.
    if(    sessionHandleType != TPM_HT_POLICY_SESSION
       || session->attributes.isAuthValueNeeded
       || session->attributes.isPasswordNeeded)
    {
        // See if entity is subject to lockout.
        if(!IsDAExempted(associatedHandle))
        {
            // If NV is unavailable, and current cycle state recorded in NV is not
            // SHUTDOWN_NONE, refuse to check any authorization because we would
            // not be able to handle a DA failure.
            result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
    }

    if(associatedHandle == TPM_RH_PLATFORM)
    {
        // If the physical presence is required for this command, check for PP
        // assertion. If it isn't asserted, no point going any further.
```



```
            if(   PhysicalPresenceIsRequired(commandCode)
               && !_plat__PhysicalPresenceAsserted()
              )
                return TPM_RC_PP;
    }
    // If a policy session is required, make sure that it is being used.
    if(   IsPolicySessionRequired(commandCode, sessionIndex)
       && sessionHandleType != TPM_HT_POLICY_SESSION)
        return TPM_RC_AUTH_TYPE;

    // If this is a PW authorization, check it and return.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
            return CheckPWAuthSession(sessionIndex);
        else
            return TPM_RC_AUTH_UNAVAILABLE;
    }
    // If this is a policy session, ...
    if(sessionHandleType == TPM_HT_POLICY_SESSION)
    {
        // ... see if the entity has a policy, ...
        if( !IsAuthPolicyAvailable(associatedHandle, commandCode, sessionIndex))
            return TPM_RC_AUTH_UNAVAILABLE;
        // ... and check the policy session.
        result = CheckPolicyAuthSession(sessionIndex, commandCode,
                                        cpHash, nameHash);
        if (result != TPM_RC_SUCCESS)
            return result;
    }
    else
    {
        // For non policy, the entity being accessed must allow authorization
        // with an auth value. This is required even if the auth value is not
        // going to be used in an HMAC because it is bound.
        if(!IsAuthValueAvailable(associatedHandle, commandCode, sessionIndex))
            return TPM_RC_AUTH_UNAVAILABLE;
    }
    // At this point, the session must be either a policy or an HMAC session.
    session = SessionGet(s_sessionHandles[sessionIndex]);

    if(     sessionHandleType == TPM_HT_POLICY_SESSION
        &&  session->attributes.isPasswordNeeded == SET)
    {
        // For policy session that requires a password, check it as PWAP session.
        return CheckPWAuthSession(sessionIndex);
    }
    else
    {
        // For other policy or HMAC sessions, have its HMAC checked.
        return CheckSessionHMAC(sessionIndex, cpHash);
    }
}
#ifdef  TPM_CC_GetCommandAuditDigest
```

7.4.4.9 CheckCommandAudit()

This function checks if the current command may trigger command audit, and if it is safe to perform the action.

Table 13

Error Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```
static TPM_RC
CheckCommandAudit(
    TPM_CC           commandCode,       // IN: Command code
    UINT32           handleNum,         // IN: number of element in handle array
    TPM_HANDLE       handles[],         // IN: array of handles
    BYTE            *parmBufferStart,   // IN: start of parameter buffer
    UINT32           parmBufferSize     // IN: size of parameter buffer
    )
{
    TPM_RC      result = TPM_RC_SUCCESS;

    // If audit is implemented, need to check to see if auditing is being done
    // for this command.
    if(CommandAuditIsRequired(commandCode))
    {
        // If the audit digest is clear and command audit is required, NV must be
        // available so that TPM2_GetCommandAuditDigest() is able to increment
        // audit counter. If NV is not available, the function bails out to prevent
        // the TPM from attempting an operation that would fail anyway.
        if(    gr.commandAuditDigest.t.size == 0
           || commandCode == TPM_CC_GetCommandAuditDigest)
        {
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
                return result;
        }
        ComputeCpHash(gp.auditHashAlg, commandCode, handleNum,
                      handles, parmBufferSize, parmBufferStart,
                      &s_cpHashForCommandAudit, NULL);
    }

    return TPM_RC_SUCCESS;
}
#endif
```

7.4.4.10 ParseSessionBuffer()

This function is the entry function for command session processing. It iterates sessions in session area and reports if the required authorization has been properly provided. It also processes audit session and passes the information of encryption sessions to parameter encryption module.

Table 14

Error Returns	Meaning
various	parsing failure or authorization failure

```
TPM_RC
ParseSessionBuffer(
    TPM_CC           commandCode,           // IN: Command code
    UINT32           handleNum,             // IN: number of element in handle array
    TPM_HANDLE       handles[],             // IN: array of handles
    BYTE            *sessionBufferStart,    // IN: start of session buffer
    UINT32           sessionBufferSize,     // IN: size of session buffer
    BYTE            *parmBufferStart,       // IN: start of parameter buffer
```

```
    UINT32               parmBufferSize         // IN: size of parameter buffer
    )
{
    TPM_RC               result;
    UINT32               i;
    INT32                size = 0;
    TPM2B_AUTH           extraKey;
    UINT32               sessionIndex;
    SESSION             *session;
    TPM2B_DIGEST         cpHash;
    TPM2B_DIGEST         nameHash;
    TPM_ALG_ID           cpHashAlg = TPM_ALG_NULL;  // algID for the last computed
                                                    // cpHash

    // Check if a command allows any session in its session area.
    if(!IsSessionAllowed(commandCode))
        return TPM_RC_AUTH_CONTEXT;

    // Default-initialization.
    s_sessionNum = 0;
    cpHash.t.size = 0;

    result = RetrieveSessionData(commandCode, &s_sessionNum,
                                 sessionBufferStart, sessionBufferSize);
    if(result != TPM_RC_SUCCESS)
        return result;

    // There is no command in the TPM spec that has more handles than
    // MAX_SESSION_NUM.
    pAssert(handleNum <= MAX_SESSION_NUM);

    // Associate the session with an authorization handle.
    for(i = 0; i < handleNum; i++)
    {
        if(CommandAuthRole(commandCode, i) != AUTH_NONE)
        {
            // If the received session number is less than the number of handle
            // that requires authorization, an error should be returned.
            // Note: for all the ISO/IEC 11889 commands, handles requiring
            // authorization come first in a command input.
            if(i > (s_sessionNum - 1))
                return TPM_RC_AUTH_MISSING;

            // Record the handle associated with the authorization session
            s_associatedHandles[i] = handles[i];
        }
    }

    // Consistency checks are done first to avoid auth failure when the command
    // will not be executed anyway.
    for(sessionIndex = 0; sessionIndex < s_sessionNum; sessionIndex++)
    {
        // PW session must be an authorization session
        if(s_sessionHandles[sessionIndex] == TPM_RS_PW )
        {
            if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
                return TPM_RC_HANDLE + g_rcIndex[sessionIndex];
        }
        else
        {
            session = SessionGet(s_sessionHandles[sessionIndex]);

            // A trial session can not appear in session area, because it cannot
            // be used for authorization, audit or encrypt/decrypt.
            if(session->attributes.isTrialPolicy == SET)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];
```

```

            // See if the session is bound to a DA protected entity
            // NOTE: Since a policy session is never bound, a policy is still
            // usable even if the object is DA protected and the TPM is in
            // lockout.
            if(session->attributes.isDaBound == SET)
            {
                result = CheckLockedOut(session->attributes.isLockoutBound == SET);
                if(result != TPM_RC_SUCCESS)
                    return result;
            }
            // If the current cpHash is the right one, don't re-compute.
            if(cpHashAlg != session->authHashAlg)   // different so compute
            {
                cpHashAlg = session->authHashAlg;   // save this new algID
                ComputeCpHash(session->authHashAlg, commandCode, handleNum,
                              handles, parmBufferSize, parmBufferStart,
                              &cpHash, &nameHash);
            }
            // If this session is for auditing, save the cpHash.
            if(s_attributes[sessionIndex].audit)
                s_cpHashForAudit = cpHash;
        }

        // if the session has an associated handle, check the auth
        if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
        {
            result = CheckAuthSession(commandCode, sessionIndex,
                                      &cpHash, &nameHash);
            if(result != TPM_RC_SUCCESS)
                return RcSafeAddToResult(result,
                                         TPM_RC_S + g_rcIndex[sessionIndex]);
        }
        else
        {
            // a session that is not for authorization must either be encrypt,
            // decrypt, or audit
            if(     s_attributes[sessionIndex].audit == CLEAR
                &&  s_attributes[sessionIndex].encrypt == CLEAR
                &&  s_attributes[sessionIndex].decrypt == CLEAR)
                return TPM_RC_ATTRIBUTES + TPM_RC_S + g_rcIndex[sessionIndex];

            // check HMAC for encrypt/decrypt/audit only sessions
            result =  CheckSessionHMAC(sessionIndex, &cpHash);
            if(result != TPM_RC_SUCCESS)
                return RcSafeAddToResult(result,
                                         TPM_RC_S + g_rcIndex[sessionIndex]);
        }
    }

#ifdef  TPM_CC_GetCommandAuditDigest
    // Check if the command should be audited.
    result = CheckCommandAudit(commandCode, handleNum, handles,
                               parmBufferStart, parmBufferSize);
    if(result != TPM_RC_SUCCESS)
        return result;              // No session number to reference
#endif

    // Decrypt the first parameter if applicable. This should be the last operation
    // in session processing.
    // If the encrypt session is associated with a handle and the handle's
    // authValue is available, then authValue is concatenated with sessionAuth to
    // generate encryption key, no matter if the handle is the session bound entity
    // or not.
    if(s_decryptSessionIndex != UNDEFINED_INDEX)
    {
```

```
            // Get size of the leading size field in decrypt parameter
            if(   s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED
               && IsAuthValueAvailable(s_associatedHandles[s_decryptSessionIndex],
                                       commandCode,
                                       s_decryptSessionIndex)
              )
            {
                extraKey.b.size=
                    EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
                                       &extraKey.t.buffer);
            }
            else
            {
                extraKey.b.size = 0;
            }
            size = DecryptSize(commandCode);
            result = CryptParameterDecryption(
                             s_sessionHandles[s_decryptSessionIndex],
                             &s_nonceCaller[s_decryptSessionIndex].b,
                             parmBufferSize, (UINT16)size,
                             &extraKey,
                             parmBufferStart);
            if(result != TPM_RC_SUCCESS)
                return RcSafeAddToResult(result,
                                         TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
        }

        return TPM_RC_SUCCESS;
    }
```

7.4.4.11 CheckAuthNoSession()

Function to process a command with no session associated. The function makes sure all the handles in the command require no authorization.

Table 15

Error Returns	Meaning
TPM_RC_AUTH_MISSING	failure - one or more handles require auth

```
TPM_RC
CheckAuthNoSession(
    TPM_CC          commandCode,        // IN: Command Code
    UINT32          handleNum,          // IN: number of handles in command
    TPM_HANDLE      handles[],          // IN: array of handles
    BYTE           *parmBufferStart,    // IN: start of parameter buffer
    UINT32          parmBufferSize      // IN: size of parameter buffer
    )
{
    UINT32 i;
    TPM_RC          result = TPM_RC_SUCCESS;

    // Check if the commandCode requires authorization
    for(i = 0; i < handleNum; i++)
    {
        if(CommandAuthRole(commandCode, i) != AUTH_NONE)
            return TPM_RC_AUTH_MISSING;
    }

#ifdef  TPM_CC_GetCommandAuditDigest
    // Check if the command should be audited.
    result = CheckCommandAudit(commandCode, handleNum, handles,
```

```
                               parmBufferStart, parmBufferSize);
    if(result != TPM_RC_SUCCESS) return result;
#endif

    // Initialize number of sessions to be 0
    s_sessionNum = 0;

    return TPM_RC_SUCCESS;
}
```

7.4.5 Response Session Processing

7.4.5.1 Introduction

The following functions build the session area in a response, and handle the audit sessions (if present).

7.4.5.2 ComputeRpHash()

Function to compute *rpHash* (Response Parameter Hash). The *rpHash* is only computed if there is an HMAC authorization session and the return code is TPM_RC_SUCCESS.

```
static void
ComputeRpHash(
    TPM_ALG_ID       hashAlg,           // IN: hash algorithm to compute rpHash
    TPM_CC           commandCode,       // IN: commandCode
    UINT32           resParmBufferSize, // IN: size of response parameter buffer
    BYTE            *resParmBuffer,     // IN: response parameter buffer
    TPM2B_DIGEST    *rpHash             // OUT: rpHash
    )
{
    // The command result in rpHash is always TPM_RC_SUCCESS.
    TPM_RC      responseCode = TPM_RC_SUCCESS;
    HASH_STATE  hashState;

    //   rpHash := hash(responseCode || commandCode || parameters)

    // Initiate hash creation.
    rpHash->t.size = CryptStartHash(hashAlg, &hashState);

    // Add hash constituents.
    CryptUpdateDigestInt(&hashState, sizeof(TPM_RC), &responseCode);
    CryptUpdateDigestInt(&hashState, sizeof(TPM_CC), &commandCode);
    CryptUpdateDigest(&hashState, resParmBufferSize, resParmBuffer);

    // Complete hash computation.
    CryptCompleteHash2B(&hashState, &rpHash->b);

    return;
}
```

7.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```
static void
InitAuditSession(
    SESSION         *session        // session to be initialized
    )
{
    // Mark session as an audit session.
```



```
    session->attributes.isAudit = SET;

    // Audit session can not be bound.
    session->attributes.isBound = CLEAR;

    // Size of the audit log is the size of session hash algorithm digest.
    session->u2.auditDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);

    // Set the original digest value to be 0.
    MemorySet(&session->u2.auditDigest.t.buffer,
              0,
              session->u2.auditDigest.t.size);

    return;
}
```

7.4.5.4 Audit()

This function updates the audit digest in an audit session.

```
static void
Audit(
    SESSION         *auditSession,      // IN: loaded audit session
    TPM_CC           commandCode,       // IN: commandCode
    UINT32           resParmBufferSize, // IN: size of response parameter buffer
    BYTE            *resParmBuffer      // IN: response parameter buffer
    )
{
    TPM2B_DIGEST     rpHash;            // rpHash for response
    HASH_STATE       hashState;

    // Compute rpHash
    ComputeRpHash(auditSession->authHashAlg,
                  commandCode,
                  resParmBufferSize,
                  resParmBuffer,
                  &rpHash);

    // auditDigestnew :=  hash (auditDigestold || cpHash || rpHash)

    // Start hash computation.
    CryptStartHash(auditSession->authHashAlg, &hashState);

    // Add old digest.
    CryptUpdateDigest2B(&hashState, &auditSession->u2.auditDigest.b);

    // Add cpHash and rpHash.
    CryptUpdateDigest2B(&hashState, &s_cpHashForAudit.b);
    CryptUpdateDigest2B(&hashState, &rpHash.b);

    // Finalize the hash.
    CryptCompleteHash2B(&hashState, &auditSession->u2.auditDigest.b);

    return;
}
#ifdef  TPM_CC_GetCommandAuditDigest
```

7.4.5.5 CommandAudit()

This function updates the command audit digest.

```
static void
CommandAudit(
```

```
    TPM_CC           commandCode,       // IN: commandCode
    UINT32           resParmBufferSize, // IN: size of response parameter buffer
    BYTE            *resParmBuffer      // IN: response parameter buffer
    )
{
    if(CommandAuditIsRequired(commandCode))
    {
        TPM2B_DIGEST    rpHash;        // rpHash for response
        HASH_STATE      hashState;

        // Compute rpHash.
        ComputeRpHash(gp.auditHashAlg, commandCode, resParmBufferSize,
                      resParmBuffer, &rpHash);

        // If the digest.size is one, it indicates the special case of changing
        // the audit hash algorithm. For this case, no audit is done on exit.
        // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
        // force an update to the NV on exit so that the change in digest will
        // be recorded. So, it is safe to exit here without setting any flags
        // because the digest change will be written to NV when this code exits.
        if(gr.commandAuditDigest.t.size == 1)
        {
            gr.commandAuditDigest.t.size = 0;
            return;
        }

        // If the digest size is zero, need to start a new digest and increment
        // the audit counter.
        if(gr.commandAuditDigest.t.size == 0)
        {
            gr.commandAuditDigest.t.size = CryptGetHashDigestSize(gp.auditHashAlg);
            MemorySet(gr.commandAuditDigest.t.buffer,
                      0,
                      gr.commandAuditDigest.t.size);

            // Bump the counter and save its value to NV.
            gp.auditCounter++;
            NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);
            g_updateNV = TRUE;
        }

        // auditDigestnew := hash (auditDigestold || cpHash || rpHash)

        //  Start hash computation.
        CryptStartHash(gp.auditHashAlg, &hashState);

        //  Add old digest.
        CryptUpdateDigest2B(&hashState, &gr.commandAuditDigest.b);

        //  Add cpHash
        CryptUpdateDigest2B(&hashState, &s_cpHashForCommandAudit.b);

        //  Add rpHash
        CryptUpdateDigest2B(&hashState, &rpHash.b);

        //  Finalize the hash.
        CryptCompleteHash2B(&hashState, &gr.commandAuditDigest.b);
    }
    return;
}
#endif
```

7.4.5.6 UpdateAuditSessionStatus()

Function to update the internal audit related states of a session. It

a) initializes the session as audit session and sets it to be exclusive if this is the first time it is used for audit or audit reset was requested;

b) reports exclusive audit session;

c) extends audit log; and

d) clears exclusive audit session if no audit session found in the command.

```
static void
UpdateAuditSessionStatus(
    TPM_CC           commandCode,       // IN: commandCode
    UINT32           resParmBufferSize, // IN: size of response parameter buffer
    BYTE            *resParmBuffer      // IN: response parameter buffer
    )
{
    UINT32           i;
    TPM_HANDLE       auditSession = TPM_RH_UNASSIGNED;

    // Iterate through sessions
    for (i = 0; i < s_sessionNum; i++)
    {
        SESSION     *session;

        // PW session do not have a loaded session and can not be an audit
        // session either.  Skip it.
        if(s_sessionHandles[i] == TPM_RS_PW) continue;

        session = SessionGet(s_sessionHandles[i]);

        // If a session is used for audit
        if(s_attributes[i].audit == SET)
        {
            // An audit session has been found
            auditSession = s_sessionHandles[i];

            // If the session has not been an audit session yet, or
            // the auditSetting bits indicate a reset, initialize it and set
            // it to be the exclusive session
            if(   session->attributes.isAudit == CLEAR
               || s_attributes[i].auditReset == SET
              )
            {
                InitAuditSession(session);
                g_exclusiveAuditSession = auditSession;
            }
            else
            {
                // Check if the audit session is the current exclusive audit
                // session and, if not, clear previous exclusive audit session.
                if(g_exclusiveAuditSession != auditSession)
                    g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
            }

            // Report audit session exclusivity.
            if(g_exclusiveAuditSession == auditSession)
            {
                s_attributes[i].auditExclusive = SET;
            }
            else
            {
                s_attributes[i].auditExclusive = CLEAR;
            }

            // Extend audit log.
            Audit(session, commandCode, resParmBufferSize, resParmBuffer);
```

```
            }
        }

        // If no audit session is found in the command, and the command allows
        // a session then, clear the current exclusive
        // audit session.
        if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(commandCode))
        {
            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
        }

        return;
    }
```

7.4.5.7 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```
    static void
    ComputeResponseHMAC(
        UINT32           sessionIndex,      // IN: session index to be processed
        SESSION         *session,           // IN: loaded session
        TPM_CC           commandCode,       // IN: commandCode
        TPM2B_NONCE     *nonceTPM,          // IN: nonceTPM
        UINT32           resParmBufferSize, // IN: size of response parameter buffer
        BYTE            *resParmBuffer,     // IN: response parameter buffer
        TPM2B_DIGEST    *hmac               // OUT: authHMAC
        )
    {
        TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
        TPM2B_KEY        key;       // HMAC key
        BYTE             marshalBuffer[sizeof(TPMA_SESSION)];
        BYTE            *buffer;
        UINT32           marshalSize;
        HMAC_STATE       hmacState;
        TPM2B_DIGEST     rp_hash;

        // Compute rpHash.
        ComputeRpHash(session->authHashAlg, commandCode, resParmBufferSize,
                      resParmBuffer, &rp_hash);

        // Generate HMAC key
        MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));

        // Check if the session has an associated handle and the associated entity is
        // the one that the session is bound to.
        // If not bound, add the authValue of this entity to the HMAC key.
        if(   s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED
           &&    !( HandleGetType(s_sessionHandles[sessionIndex])
                      == TPM_HT_POLICY_SESSION
              &&   session->attributes.isAuthValueNeeded == CLEAR)
           && !session->attributes.requestWasBound)
        {
            pAssert((sizeof(AUTH_VALUE) + key.t.size) <= <K>sizeof(key.t.buffer));
            key.t.size = key.t.size +
                             EntityGetAuthValue(s_associatedHandles[sessionIndex],
                                                  (AUTH_VALUE *)&key.t.buffer[key.t.size]);
        }

        // if the HMAC key size for a policy session is 0, the response HMAC is
        // computed according to the input HMAC
        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
            && key.t.size == 0
            && s_inputAuthValues[sessionIndex].t.size == 0)
```



```
    {
        hmac->t.size = 0;
        return;
    }

    // Start HMAC computation.
    hmac->t.size = CryptStartHMAC2B(session->authHashAlg, &key.b, &hmacState);

    // Add hash components.
    CryptUpdateDigest2B(&hmacState, &rp_hash.b);
    CryptUpdateDigest2B(&hmacState, &nonceTPM->b);
    CryptUpdateDigest2B(&hmacState, &s_nonceCaller[sessionIndex].b);

    // Add session attributes.
    buffer = marshalBuffer;
    marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
    CryptUpdateDigest(&hmacState, marshalSize, marshalBuffer);

    // Finalize HMAC.
    CryptCompleteHMAC2B(&hmacState, &hmac->b);

    return;
}
```

7.4.5.8 BuildSingleResponseAuth()

Function to compute response for an authorization session.

```
static void
BuildSingleResponseAuth(
    UINT32           sessionIndex,      // IN: session index to be processed
    TPM_CC           commandCode,       // IN: commandCode
    UINT32           resParmBufferSize, // IN: size of response parameter buffer
    BYTE            *resParmBuffer,     // IN: response parameter buffer
    TPM2B_AUTH      *auth               // OUT: authHMAC
    )
{
    // For password authorization, field is empty.
    if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
    {
        auth->t.size = 0;
    }
    else
    {
        // Fill in policy/HMAC based session response.
        SESSION     *session = SessionGet(s_sessionHandles[sessionIndex]);

        // If the session is a policy session with isPasswordNeeded SET, the auth
        // field is empty.
        if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
                && session->attributes.isPasswordNeeded == SET)
            auth->t.size = 0;
        else
            // Compute response HMAC.
            ComputeResponseHMAC(sessionIndex,
                                session,
                                commandCode,
                                &session->nonceTPM,
                                resParmBufferSize,
                                resParmBuffer,
                                auth);
    }

    return;
```

```
}
```

7.4.5.9 UpdateTPMNonce()

Updates TPM nonce in both internal session or response if applicable.

```
static void
UpdateTPMNonce(
    UINT16          noncesSize,     // IN: number of elements in 'nonces' array
    TPM2B_NONCE     nonces[]        // OUT: nonceTPM
    )
{
    UINT32      i;
    pAssert(noncesSize >= s_sessionNum);
    for(i = 0; i < s_sessionNum; i++)
    {
        SESSION     *session;
        // For PW session, nonce is 0.
        if(s_sessionHandles[i] == TPM_RS_PW)
        {
            nonces[i].t.size = 0;
            continue;
        }
        session = SessionGet(s_sessionHandles[i]);
        // Update nonceTPM in both internal session and response.
        CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
        nonces[i] = session->nonceTPM;
    }
    return;
}
```

7.4.5.10 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

```
static void
UpdateInternalSession(
    void
    )
{
    UINT32      i;
    for(i = 0; i < s_sessionNum; i++)
    {
        // For PW session, no update.
        if(s_sessionHandles[i] == TPM_RS_PW) continue;

        if(s_attributes[i].continueSession == CLEAR)
        {
            // Close internal session.
            SessionFlush(s_sessionHandles[i]);
        }
        else
        {
            // If nonce is rolling in a policy session, the policy related data
            // will be re-initialized.
            if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION)
            {
                SESSION     *session = SessionGet(s_sessionHandles[i]);

```



```
                // When the nonce rolls it starts a new timing interval for the
                // policy session.
                SessionResetPolicyData(session);
                session->startTime = go.clock;
            }
        }
    }
    return;
}
```

7.4.5.11 BuildResponseSession()

Function to build Session buffer in a response.

```
void
BuildResponseSession(
    TPM_ST           tag,            // IN: tag
    TPM_CC           commandCode,    // IN: commandCode
    UINT32           resHandleSize,  // IN: size of response handle buffer
    UINT32           resParmSize,    // IN: size of response parameter buffer
    UINT32          *resSessionSize  // OUT: response session area
    )
{
    BYTE            *resParmBuffer;
    TPM2B_NONCE  responseNonces[MAX_SESSION_NUM];

    // Compute response parameter buffer start.
    resParmBuffer = MemoryGetResponseBuffer(commandCode) + sizeof(TPM_ST) +
                    sizeof(UINT32) + sizeof(TPM_RC) + resHandleSize;

    // For TPM_ST_SESSIONS, there is parameterSize field.
    if(tag == TPM_ST_SESSIONS)
        resParmBuffer += sizeof(UINT32);

    // Session nonce should be updated before parameter encryption
    if(tag == TPM_ST_SESSIONS)
    {
        UpdateTPMNonce(MAX_SESSION_NUM, responseNonces);

        // Encrypt first parameter if applicable. Parameter encryption should
        // happen after nonce update and before any rpHash is computed.
        // If the encrypt session is associated with a handle, the authValue of
        // this handle will be concatenated with sessionAuth to generate
        // encryption key, no matter if the handle is the session bound entity
        // or not. The authValue is added to sessionAuth only when the authValue
        // is available.
        if(s_encryptSessionIndex != UNDEFINED_INDEX)
        {
            UINT32          size;
            TPM2B_AUTH      extraKey;

            // Get size of the leading size field
            if(   s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED
               && IsAuthValueAvailable(s_associatedHandles[s_encryptSessionIndex],
                                       commandCode, s_encryptSessionIndex)
              )
            {
                extraKey.b.size =
                    EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
                                       &extraKey.t.buffer);
            }
            else
            {
                extraKey.b.size = 0;
```

```
                }
                size = EncryptSize(commandCode);
                CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
                                         &s_nonceCaller[s_encryptSessionIndex].b,
                                         (UINT16)size,
                                         &extraKey,
                                         resParmBuffer);

            }

        }
        // Audit session should be updated first regardless of the tag.
        // A command with no session may trigger a change of the exclusivity state.
        UpdateAuditSessionStatus(commandCode, resParmSize, resParmBuffer);

        // Audit command.
        CommandAudit(commandCode, resParmSize, resParmBuffer);

        // Process command with sessions.
        if(tag == TPM_ST_SESSIONS)
        {
            UINT32          i;
            BYTE           *buffer;
            TPM2B_DIGEST    responseAuths[MAX_SESSION_NUM];

            pAssert(s_sessionNum > 0);

            // Iterate over each session in the command session area, and create
            // corresponding sessions for response.
            for(i = 0; i < s_sessionNum; i++)
            {
                BuildSingleResponseAuth(
                                        i,
                                        commandCode,
                                        resParmSize,
                                        resParmBuffer,
                                        &responseAuths[i]);
                // Make sure that continueSession is SET on any Password session.
                // This makes it marginally easier for the management software
                // to keep track of the closed sessions.
                if(   s_attributes[i].continueSession == CLEAR
                   && s_sessionHandles[i] == TPM_RS_PW)
                {
                    s_attributes[i].continueSession = SET;
                }
            }

            // Assemble Response Sessions.
            *resSessionSize = 0;
            buffer = resParmBuffer + resParmSize;
            for(i = 0; i < s_sessionNum; i++)
            {
                *resSessionSize += TPM2B_NONCE_Marshal(&responseNonces[i],
                                                       &buffer, NULL);
                *resSessionSize += TPMA_SESSION_Marshal(&s_attributes[i],
                                                        &buffer, NULL);
                *resSessionSize += TPM2B_DIGEST_Marshal(&responseAuths[i],
                                                        &buffer, NULL);
            }

            // Update internal sessions after completing response buffer computation.
            UpdateInternalSession();
        }
        else
        {
            // Process command with no session.
```

```
            *resSessionSize = 0;
        }

        return;
    }
```

8 Command Support Functions

8.1 Introduction

Clause 8 contains support routines that are called by the command action code in ISO/IEC 11889-3. The functions are grouped by the command group that is supported by the functions.

8.2 Attestation Command Support (Attest_spt.c)

8.2.1 Includes

```
#include "InternalRoutines.h"
#include "Attest_spt_fp.h"
```

8.2.2 Functions

8.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

Table 16

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*

```
TPM_RC
FillInAttestInfo(
    TPMI_DH_OBJECT       signHandle,    // IN: handle of signing object
    TPMT_SIG_SCHEME     *scheme,        // IN/OUT: scheme to be used for signing
    TPM2B_DATA          *data,          // IN: qualifying data
    TPMS_ATTEST         *attest         // OUT: attest structure
    )
{
    TPM_RC                  result;
    TPMI_RH_HIERARCHY       signHierarhcy;

    result = CryptSelectSignScheme(signHandle, scheme);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Magic number
    attest->magic = TPM_GENERATED_VALUE;

    if(signHandle == TPM_RH_NULL)
    {
        BYTE    *buffer;
        // For null sign handle, the QN is TPM_RH_NULL
        buffer = attest->qualifiedSigner.t.name;
        attest->qualifiedSigner.t.size =
            TPM_HANDLE_Marshal(&signHandle, &buffer, NULL);
    }
    else
    {
        // Certifying object qualified name
```



```
        // if the scheme is anonymous, this is an empty buffer
        if(CryptIsSchemeAnonymous(scheme->scheme))
            attest->qualifiedSigner.t.size = 0;
        else
            ObjectGetQualifiedName(signHandle, &attest->qualifiedSigner);
    }

    // current clock in plain text
    TimeFillInfo(&attest->clockInfo);

    // Firmware version in plain text
    attest->firmwareVersion = ((UINT64) gp.firmwareV1 << (<K>sizeof(UINT32) * 8));
    attest->firmwareVersion += gp.firmwareV2;

    // Get the hierarchy of sign object.  For NULL sign handle, the hierarchy
    // will be TPM_RH_NULL
    signHierarhcy = EntityGetHierarchy(signHandle);
    if(signHierarhcy != TPM_RH_PLATFORM && signHierarhcy != TPM_RH_ENDORSEMENT)
    {
        // For sign object is not in platform or endorsement hierarchy,
        // obfuscate the clock and firmwereVersion information
        UINT64          obfuscation[2];
        TPMI_ALG_HASH   hashAlg;

        // Get hash algorithm
        if(signHandle == TPM_RH_NULL || signHandle == TPM_RH_OWNER)
        {
            hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
        }
        else
        {
            OBJECT          *signObject = NULL;
            signObject = ObjectGet(signHandle);
            hashAlg = signObject->publicArea.nameAlg;
        }
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"for normative KDF
        // label values.
        KDFa(hashAlg, &gp.shProof.b, "OBFUSCATE",
             &attest->qualifiedSigner.b, NULL, 128, (BYTE *)&obfuscation[0], NULL);

        // Obfuscate data
        attest->firmwareVersion += obfuscation[0];
        attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
        attest->clockInfo.restartCount += (UINT32)obfuscation[1];
    }

    // External data
    if(CryptIsSchemeAnonymous(scheme->scheme))
        attest->extraData.t.size = 0;
    else
    {
        // If we move the data to the attestation structure, then we will not use
        // it in the signing operation except as part of the signed data
        attest->extraData = *data;
        data->t.size = 0;
    }

    return TPM_RC_SUCCESS;
}
```

8.2.2.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If *signHandle* is TPM_RH_NULL, a null signature is returned.

Table 17

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*signHandle* references not a signing key
TPM_RC_SCHEME	*scheme* is not compatible with *signHandle* type
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
SignAttestInfo(
    TPMI_DH_OBJECT       signHandle,        // IN: handle of sign object
    TPMT_SIG_SCHEME     *scheme,            // IN: sign scheme
    TPMS_ATTEST         *certifyInfo,       // IN: the data to be signed
    TPM2B_DATA          *qualifyingData,    // IN: extra data for the signing
                                            //     process
    TPM2B_ATTEST        *attest,            // OUT: marshaled attest blob to be
                                            //     signed
    TPMT_SIGNATURE      *signature          // OUT: signature
    )
{
    TPM_RC                  result;
    TPMI_ALG_HASH           hashAlg;
    BYTE                    *buffer;
    HASH_STATE              hashState;
    TPM2B_DIGEST            digest;


    // Marshal TPMS_ATTEST structure for hash
    buffer = attest->t.attestationData;
    attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);

    if(signHandle == TPM_RH_NULL)
    {
        signature->sigAlg = TPM_ALG_NULL;
    }
    else
    {
        // Attestation command may cause the orderlyState to be cleared due to
        // the reporting of clock info.  If this is the case, check if NV is
        // available first.
        if(gp.orderlyState != SHUTDOWN_NONE)
        {
            // The command needs NV update.  Check if NV is available.
            // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
            // this point
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
                return result;
        }

        // Compute hash
        hashAlg = scheme->details.any.hashAlg;
        digest.t.size = CryptStartHash(hashAlg, &hashState);
        CryptUpdateDigest(&hashState, attest->t.size, attest->t.attestationData);
        CryptCompleteHash2B(&hashState, &digest.b);

        // If there is qualifying data, need to rehash the the data
        // hash(qualifyingData || hash(attestationData))
        if(qualifyingData->t.size != 0)
        {
            CryptStartHash(hashAlg, &hashState);
            CryptUpdateDigest(&hashState,
```



```
                            qualifyingData->t.size,
                            qualifyingData->t.buffer);
            CryptUpdateDigest(&hashState, digest.t.size, digest.t.buffer);
            CryptCompleteHash2B(&hashState, &digest.b);
        }

        // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
        // TPM_RC_ATTRIBUTES error may be returned at this point
        return CryptSign(signHandle,
                         scheme,
                         &digest,
                         signature);
    }

    return TPM_RC_SUCCESS;
}
```

8.3 Context Management Command Support (Context_spt.c)

8.3.1 Includes

```
#include "InternalRoutines.h"
#include "Context_spt_fp.h"
```

8.3.2 Functions

8.3.2.1 ComputeContextProtectionKey()

This function retrieves the symmetric protection key for context encryption It is used by TPM2_ConextSave() and TPM2_ContextLoad() to create the symmetric encryption key and iv

```
void
ComputeContextProtectionKey(
    TPMS_CONTEXT    *contextBlob,   // IN: context blob
    TPM2B_SYM_KEY   *symKey,        // OUT: the symmetric key
    TPM2B_IV        *iv             // OUT: the IV.
    )
{
    UINT16           symKeyBits;    // number of bits in the parent's
                                    //   symmetric key
    TPM2B_AUTH      *proof = NULL;  // the proof value to use. Is null for
                                    //   everything but a primary object in
                                    //   the Endorsement Hierarchy

    BYTE             kdfResult[sizeof(TPMU_HA) * 2];// Value produced by the KDF

    TPM2B_DATA       sequence2B, handle2B;

    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);

    // Get sequence value in 2B format
    sequence2B.t.size = sizeof(contextBlob->sequence);
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
               sizeof(contextBlob->sequence),
               sizeof(sequence2B.t.buffer));

    // Get handle value in 2B format
    handle2B.t.size = sizeof(contextBlob->savedHandle);
    MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
               sizeof(contextBlob->savedHandle),
```

```
                 sizeof(handle2B.t.buffer));

    // Get the symmetric encryption key size
    symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
    symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
    // Get the size of the IV for the algorithm
    iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);

    // KDFa to generate symmetric key and IV value
    KDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, "CONTEXT", &sequence2B.b,
         &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL);

    // Copy part of the returned value as the key
    MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size,
               sizeof(symKey->t.buffer));

    // Copy the rest as the IV
    MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size,
               sizeof(iv->t.buffer));

    return;
}
```

8.3.2.2 ComputeContextIntegrity()

Generate the integrity hash for a context It is used by TPM2_ContextSave() to create an integrity hash and by TPM2_ContextLoad() to compare an integrity hash

```
void
ComputeContextIntegrity(
    TPMS_CONTEXT    *contextBlob,   // IN: context blob
    TPM2B_DIGEST    *integrity      // OUT: integrity
    )
{
    HMAC_STATE          hmacState;
    TPM2B_AUTH          *proof;
    UINT16              integritySize;

    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);

    // Start HMAC
    integrity->t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                          &proof->b, &hmacState);

    // Compute integrity size at the beginning of context blob
    integritySize = sizeof(integrity->t.size) + integrity->t.size;


    // Adding total reset counter so that the context cannot be
    // used after a TPM Reset
    CryptUpdateDigestInt(&hmacState, sizeof(gp.totalResetCount),
                         &gp.totalResetCount);

    // If this is a ST_CLEAR object, add the clear count
    // so that this contest cannot be loaded after a TPM Restart
    if(contextBlob->savedHandle == 0x80000002)
        CryptUpdateDigestInt(&hmacState, sizeof(gr.clearCount), &gr.clearCount);

    // Adding sequence number to the HMAC to make sure that it doesn't
    // get changed
    CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->sequence),
                         &contextBlob->sequence);

```



```
    // Protect the handle
    CryptUpdateDigestInt(&hmacState, sizeof(contextBlob->savedHandle),
                         &contextBlob->savedHandle);

    // Adding sensitive contextData, skip the leading integrity area
    CryptUpdateDigest(&hmacState, contextBlob->contextBlob.t.size - integritySize,
                      contextBlob->contextBlob.t.buffer + integritySize);

    // Complete HMAC
    CryptCompleteHMAC2B(&hmacState, &integrity->b);

    return;
}
```

8.3.2.3 SequenceDataImportExport()

This function is used scan through the sequence object and either modify the hash state data for LIB_EXPORT or to import it into the internal format

```
void
SequenceDataImportExport(
    OBJECT          *object,        // IN: the object containing the sequence data
    OBJECT          *exportObject,  // IN/OUT: the object structure that will get
                                    //      the exported hash state
    IMPORT_EXPORT    direction
    )
{
    int                     count = 1;
    HASH_OBJECT            *internalFmt = (HASH_OBJECT *)object;
    HASH_OBJECT            *externalFmt = (HASH_OBJECT *)exportObject;


    if(object->attributes.eventSeq)
        count = HASH_COUNT;
    for(; count; count--)
        CryptHashStateImportExport(&internalFmt->state.hashState[count - 1],
                              externalFmt->state.hashState, direction);
}
```

8.4 Policy Command Support (Policy_spt.c)

8.4.1 Includes

```
#include "InternalRoutines.h"
#include "Policy_spt_fp.h"
#include "PolicySigned_fp.h"
#include "PolicySecret_fp.h"
#include "PolicyTicket_fp.h"
```

8.4.2 Functions

8.4.2.1 PolicyParameterChecks()

This function validates the common parameters of TPM2_PolicySiged() and TPM2_PolicySecret(). The common parameters are *nonceTPM*, *expiration*, and *cpHashA*.

```
TPM_RC
PolicyParameterChecks(
    SESSION         *session,
    UINT64           authTimeout,
```

```
    TPM2B_DIGEST    *cpHashA,
    TPM2B_NONCE     *nonce,
    TPM_RC           nonceParameterNumber,
    TPM_RC           cpHashParameterNumber,
    TPM_RC           expirationParameterNumber
    )
{
    TPM_RC           result;

    // Validate that input nonceTPM is correct if present
    if(nonce != NULL && nonce->t.size != 0)
    {
        if(!Memory2BEqual(&nonce->b, &session->nonceTPM.b))
            return TPM_RC_NONCE + RC_PolicySigned_nonceTPM;
    }
    // If authTimeout is set (expiration != 0...
    if(authTimeout != 0)
    {
        // ...then nonce must be present
        // nonce present isn't checked in PolicyTicket
        if(nonce != NULL && nonce->t.size == 0)
            // This error says that the time has expired but it is pointing
            // at the nonceTPM value.
            return TPM_RC_EXPIRED + nonceParameterNumber;

        // Validate input expiration.
        // Cannot compare time if clock stop advancing.  A TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;

        if(authTimeout < go.clock)
            return TPM_RC_EXPIRED + expirationParameterNumber;
    }
    // If the cpHash is present, then check it
    if(cpHashA != NULL && cpHashA->t.size != 0)
    {
        // The cpHash input has to have the correct size
        if(cpHashA->t.size != session->u2.policyDigest.t.size)
            return TPM_RC_SIZE + cpHashParameterNumber;

        // If the cpHash has already been set, then this input value
        // must match the current value.
        if(     session->u1.cpHash.b.size != 0
            &&  !Memory2BEqual(&cpHashA->b, &session->u1.cpHash.b))
                return TPM_RC_CPHASH;
    }
    return TPM_RC_SUCCESS;
}
```

8.4.2.2 PolicyContextUpdate()

Update policy hash Update the *policyDigest* in policy session by extending *policyRef* and *objectName* to it. This will also update the *cpHash* if it is present.

```
void
PolicyContextUpdate(
    TPM_CC           commandCode,    // IN: command code
    TPM2B_NAME      *name,           // IN: name of entity
    TPM2B_NONCE     *ref,            // IN: the reference data
    TPM2B_DIGEST    *cpHash,         // IN: the cpHash (optional)
    UINT64           policyTimeout,
    SESSION         *session         // IN/OUT: policy session to be updated
```



```
    )
{
    HASH_STATE          hashState;
    UINT16              policyDigestSize;

    // Start hash
    policyDigestSize = CryptStartHash(session->authHashAlg, &hashState);

    // policyDigest size should always be the digest size of session hash algorithm.
    pAssert(session->u2.policyDigest.t.size == policyDigestSize);

    // add old digest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);

    // add commandCode
    CryptUpdateDigestInt(&hashState, sizeof(commandCode), &commandCode);

    // add name if applicable
    if(name != NULL)
        CryptUpdateDigest2B(&hashState, &name->b);

    // Complete the digest and get the results
    CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);

    // Start second hash computation
    CryptStartHash(session->authHashAlg, &hashState);

    // add policyDigest
    CryptUpdateDigest2B(&hashState, &session->u2.policyDigest.b);

    // add policyRef
    if(ref != NULL)
        CryptUpdateDigest2B(&hashState, &ref->b);

    // Complete second digest
    CryptCompleteHash2B(&hashState, &session->u2.policyDigest.b);

    // Deal with the cpHash. If the cpHash value is present
    // then it would have already been checked to make sure that
    // it is compatible with the current value so all we need
    // to do here is copy it and set the iscoHashDefined attribute
    if(cpHash != NULL && cpHash->t.size != 0)
    {
        session->u1.cpHash = *cpHash;
        session->attributes.iscpHashDefined = SET;
    }

    // update the timeout if it is specified
    if(policyTimeout!= 0)
    {
    // If the timeout has not been set, then set it to the new value
        if(session->timeOut == 0)
            session->timeOut = policyTimeout;
        else if(session->timeOut > policyTimeout)
            session->timeOut =  policyTimeout;
    }
    return;
}
```

8.5 NV Command Support (NV_spt.c)

8.5.1 Includes

```
#include "InternalRoutines.h"
```

```
#include "NV_spt_fp.h"
```

8.5.2 Functions

8.5.2.1 NvReadAccessChecks()

Common routine for validating a read Used by TPM2_NV_Read(), TPM2_NV_ReadLock() and TPM2_PolicyNV().

Table 18

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	*authHandle* is not allowed to authorize read of the index
TPM_RC_NV_LOCKED	Read locked
TPM_RC_NV_UNINITIALIZED	Try to read an uninitialized index

```
TPM_RC
NvReadAccessChecks(
    TPM_HANDLE      authHandle,     // IN: the handle that provided the
                                    //     authorization
    TPM_HANDLE      nvHandle        // IN: the handle of the NV index to be written
    )
{
    NV_INDEX        nvIndex;

    // Get NV index info
    NvGetIndexInfo(nvHandle, &nvIndex);

// This check may be done before doing authorization checks as is done in this
// version of the reference code. If not done there, then uncomment the next
// three lines.
//    // If data is read locked, returns an error
//    if(nvIndex.publicArea.attributes.TPMA_NV_READLOCKED == SET)
//        return TPM_RC_NV_LOCKED;

    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the read.  If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked..
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided auth then ONWERWRITE must be SET
        if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided auth then PPWRITE must be SET
        if(!nvIndex.publicArea.attributes.TPMA_NV_PPREAD)
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided auth, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
            return TPM_RC_NV_AUTHORIZATION;

    // If the index has not been written, then the value cannot be read
    // NOTE: This has to come after other access checks to make sure that
    // the proper authorization is given to TPM2_NV_ReadLock()
    if(nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
```



```
            return TPM_RC_NV_UNINITIALIZED;

        return TPM_RC_SUCCESS;
}
```

8.5.2.2 NvWriteAccessChecks()

Common routine for validating a write Used by TPM2_NV_Write(), TPM2_NV_Increment(), TPM2_SetBits(), and TPM2_NV_WriteLock().

Table 19

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```
TPM_RC
NvWriteAccessChecks(
    TPM_HANDLE      authHandle,    // IN: the handle that provided the
                                   //     authorization
    TPM_HANDLE      nvHandle       // IN: the handle of the NV index to be written
    )
{
    NV_INDEX        nvIndex;

    // Get NV index info
    NvGetIndexInfo(nvHandle, &nvIndex);

// This check may be done before doing authorization checks as is done in this
// version of the reference code. If not done there, then uncomment the next
// three lines.
//    // If data is write locked, returns an error
//    if(nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED == SET)
//        return TPM_RC_NV_LOCKED;

    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the write.  If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked..
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided auth then ONWERWRITE must be SET
        if(! nvIndex.publicArea.attributes.TPMA_NV_OWNERWRITE)
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided auth then PPWRITE must be SET
        if(!nvIndex.publicArea.attributes.TPMA_NV_PPWRITE)
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided auth, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
            return TPM_RC_NV_AUTHORIZATION;

    return TPM_RC_SUCCESS;
}
```

8.6 Object Command Support (Object_spt.c)

8.6.1 Includes

```
#include "InternalRoutines.h"
#include "Object_spt_fp.h"
#include <Platform.h>
```

8.6.2 Local Functions

8.6.2.1 EqualCryptSet()

Check if the crypto sets in two public areas are equal.

Table 20

Error Returns	Meaning
TPM_RC_ASYMMETRIC	mismatched parameters
TPM_RC_HASH	mismatched name algorithm
TPM_RC_TYPE	mismatched type

```
static TPM_RC
EqualCryptSet(
    TPMT_PUBLIC     *publicArea1,    // IN: public area 1
    TPMT_PUBLIC     *publicArea2     // IN: public area 2
    )
{
    UINT16              size1;
    UINT16              size2;
    BYTE                params1[sizeof(TPMU_PUBLIC_PARMS)];
    BYTE                params2[sizeof(TPMU_PUBLIC_PARMS)];
    BYTE                *buffer;

    // Compare name hash
    if(publicArea1->nameAlg != publicArea2->nameAlg)
        return TPM_RC_HASH;

    // Compare algorithm
    if(publicArea1->type != publicArea2->type)
        return TPM_RC_TYPE;

    // TPMU_PUBLIC_PARMS field should be identical
    buffer = params1;
    size1 = TPMU_PUBLIC_PARMS_Marshal(&publicArea1->parameters, &buffer,
                                      NULL, publicArea1->type);
    buffer = params2;
    size2 = TPMU_PUBLIC_PARMS_Marshal(&publicArea2->parameters, &buffer,
                                      NULL, publicArea2->type);

    if(size1 != size2 || !MemoryEqual(params1, params2, size1))
        return TPM_RC_ASYMMETRIC;

    return TPM_RC_SUCCESS;
}
```


8.6.2.2 GetIV2BSize()

Get the size of TPM2B_IV in canonical form that will be append to the start of the sensitive data. It includes both size of size field and size of iv data.

```
static UINT16
GetIV2BSize(
    TPM_HANDLE          protectorHandle    // IN: the protector handle
    )
{
    OBJECT              *protector = NULL; // Pointer to the protector object
    TPM_ALG_ID          symAlg;
    UINT16              keyBits;

    // Determine the symmetric algorithm and size of key
    if(protectorHandle == TPM_RH_NULL)
    {
        // Use the context encryption algorithm and key size
        symAlg = CONTEXT_ENCRYPT_ALG;
        keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        protector = ObjectGet(protectorHandle);
        symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
        keyBits= protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
    }

    // The IV size is a UINT16 size field plus the block size of the symmetric
    // algorithm
    return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
}
```

8.6.2.3 ComputeProtectionKeyParms()

This function retrieves the symmetric protection key parameters for the sensitive data The parameters retrieved from this function include encryption algorithm, key size in bit, and a TPM2B_SYM_KEY containing the key material as well as the key size in bytes This function is used for any action that requires encrypting or decrypting of the sensitive area of an object or a credential blob.

```
static void
ComputeProtectionKeyParms(
    TPM_HANDLE          protectorHandle,   // IN: the protector handle
    TPM_ALG_ID          hashAlg,           // IN: hash algorithm for KDFa
    TPM2B_NAME         *name,              // IN: name of the object
    TPM2B_SEED         *seedIn,            // IN: optional seed for duplication blob.
                                           //     For non duplication blob, this
                                           //     parameter should be NULL
    TPM_ALG_ID         *symAlg,            // OUT: the symmetric algorithm
    UINT16             *keyBits,           // OUT: the symmetric key size in bits
    TPM2B_SYM_KEY      *symKey             // OUT: the symmetric key
    )
{
    TPM2B_SEED          *seed = NULL;
    OBJECT              *protector = NULL; // Pointer to the protector

    // Determine the algorithms for the KDF and the encryption/decryption
    // For TPM_RH_NULL, using context settings
    if(protectorHandle == TPM_RH_NULL)
    {
        // Use the context encryption algorithm and key size
        *symAlg = CONTEXT_ENCRYPT_ALG;
        symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
```

```
        *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    }
    else
    {
        TPMT_SYM_DEF_OBJECT *symDef;
        protector = ObjectGet(protectorHandle);
        symDef = &protector->publicArea.parameters.asymDetail.symmetric;
        *symAlg = symDef->algorithm;
        *keyBits= symDef->keyBits.sym;
        symKey->t.size = (*keyBits + 7) / 8;
    }

    // Get seed for KDF
    seed = GetSeedForKDF(protectorHandle, seedIn);

    // KDFa to generate symmetric key and IV value
    // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"
    KDFa(hashAlg, (TPM2B *)seed, "STORAGE", (TPM2B *)name, NULL,
         symKey->t.size * 8, symKey->t.buffer, NULL);

    return;
}
```

8.6.2.4 ComputeOuterIntegrity()

The sensitive area parameter is a buffer that holds a space for the integrity value and the marshaled sensitive area. The caller should skip over the area set aside for the integrity value and compute the hash of the remainder of the object. The size field of sensitive is in unmarshaled form and the sensitive area contents is an array of bytes.

```
static void
ComputeOuterIntegrity(
    TPM2B_NAME      *name,              // IN: the name of the object
    TPM_HANDLE       protectorHandle,   // IN: The handle of the object that
                                        //     provides protection.  For object, it
                                        //     is parent handle. For credential, it
                                        //     is the handle of encrypt object.  For
                                        //     a Temporary Object, it is TPM_RH_NULL
    TPMI_ALG_HASH    hashAlg,           // IN: algorithm to use for integrity
    TPM2B_SEED      *seedIn,            // IN: an external seed may be provided for
                                        //     duplication blob. For non duplication
                                        //     blob, this parameter should be NULL
    UINT32           sensitiveSize,     // IN: size of the marshaled sensitive data
    BYTE            *sensitiveData,     // IN: sensitive area
    TPM2B_DIGEST    *integrity          // OUT: integrity
    )
{
    HMAC_STATE           hmacState;

    TPM2B_DIGEST         hmacKey;
    TPM2B_SEED          *seed = NULL;

    // Get seed for KDF
    seed = GetSeedForKDF(protectorHandle, seedIn);

    // Determine the HMAC key bits
    hmacKey.t.size = CryptGetHashDigestSize(hashAlg);

    // KDFa to generate HMAC key
    // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters"
    KDFa(hashAlg, (TPM2B *)seed, "INTEGRITY", NULL, NULL,
         hmacKey.t.size * 8, hmacKey.t.buffer, NULL);

    // Start HMAC and get the size of the digest which will become the integrity
```



```
    integrity->t.size = CryptStartHMAC2B(hashAlg, &hmacKey.b, &hmacState);

    // Adding the marshaled sensitive area to the integrity value
    CryptUpdateDigest(&hmacState, sensitiveSize, sensitiveData);

    // Adding name
    CryptUpdateDigest2B(&hmacState, (TPM2B *)name);

    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &integrity->b);

    return;
}
```

8.6.2.5 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap.

```
static void
ComputeInnerIntegrity(
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
    TPM2B_NAME      *name,          // IN: the name of the object
    UINT16           dataSize,      // IN: the size of sensitive data
    BYTE            *sensitiveData, // IN: sensitive data
    TPM2B_DIGEST    *integrity      // OUT: inner integrity
    )
{
    HASH_STATE      hashState;

    // Start hash and get the size of the digest which will become the integrity
    integrity->t.size = CryptStartHash(hashAlg, &hashState);

    // Adding the marshaled sensitive area to the integrity value
    CryptUpdateDigest(&hashState, dataSize, sensitiveData);

    // Adding name
    CryptUpdateDigest2B(&hashState, &name->b);

    // Compute hash
    CryptCompleteHash2B(&hashState, &integrity->b);

    return;

}
```

8.6.2.6 ProduceInnerIntegrity()

This function produces an inner integrity for regular private, credential or duplication blob It requires the sensitive data being marshaled to the *innerBuffer*, with the leading bytes reserved for integrity hash. It assume the sensitive data starts at address *(innerBuffer* + integrity size). This function integrity at the beginning of the inner buffer It returns the total size of buffer with the inner wrap.

```
static UINT16
ProduceInnerIntegrity(
    TPM2B_NAME      *name,          // IN: the name of the object
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
    UINT16           dataSize,      // IN: the size of sensitive data, excluding the
                                    //     leading integrity buffer size
    BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
                                    //     it.  At input, the leading bytes of this
                                    //     buffer is reserved for integrity
    )
```

```
{
    BYTE                *sensitiveData; // pointer to the sensitive data

    TPM2B_DIGEST        integrity;
    UINT16              integritySize;
    BYTE                *buffer;        // Auxiliary buffer pointer

    // sensitiveData points to the beginning of sensitive data in innerBuffer
    integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
    sensitiveData = innerBuffer + integritySize;

    ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);

    // Add integrity at the beginning of inner buffer
    buffer = innerBuffer;
    TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);

    return dataSize + integritySize;
}
```

8.6.2.7 CheckInnerIntegrity()

This function check integrity of inner blob.

Table 21

Error Returns	Meaning
TPM_RC_INTEGRITY	if the outer blob integrity is bad
unmarshal errors	unmarshal errors while unmarshaling integrity

```
static TPM_RC
CheckInnerIntegrity(
    TPM2B_NAME      *name,          // IN: the name of the object
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for inner wrap
    UINT16           dataSize,      // IN: the size of sensitive data, including the
                                    //     leading integrity buffer size
    BYTE            *innerBuffer    // IN/OUT: inner buffer with sensitive data in
                                    //     it
    )
{
    TPM_RC          result;

    TPM2B_DIGEST    integrity;
    TPM2B_DIGEST    integrityToCompare;
    BYTE            *buffer;                // Auxiliary buffer pointer
    INT32           size;

    // Unmarshal integrity
    buffer = innerBuffer;
    size = (INT32) dataSize;
    result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
    if(result == TPM_RC_SUCCESS)
    {
        // Compute integrity to compare
        ComputeInnerIntegrity(hashAlg, name, (UINT16) size, buffer,
                              &integrityToCompare);

        // Compare outer blob integrity
        if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
            result = TPM_RC_INTEGRITY;
    }
    return result;
```



```
}
```

8.6.3 Public Functions

8.6.3.1 AreAttributesForParent()

This function is called by create, load, and import functions.

Table 22

Return Value	Meaning
TRUE	properties are those of a parent
FALSE	properties are not those of a parent

```
BOOL
AreAttributesForParent(
    OBJECT          *parentObject   // IN: parent handle
    )
{
    // This function is only called when a parent is needed. Any
    // time a "parent" is used, it must be authorized. When
    // the authorization is checked, both the public and sensitive
    // areas must be loaded. Just make sure...
    pAssert(parentObject->attributes.publicOnly == CLEAR);


    if(ObjectDataIsStorage(&parentObject->publicArea))
        return TRUE;
    else
        return FALSE;
}
```

8.6.3.2 SchemeChecks()

This function validates the schemes in the public area of an object. This function is called by TPM2_LoadExternal() and PublicAttributesValidation().

Table 23

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
TPM_RC_ATTRIBUTES	attempt to inject sensitive data for an asymmetric key; or attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unexpected object type; or non-duplicable storage key and its parent have different types


```
TPM_RC
SchemeChecks(
    BOOL              load,           // IN: TRUE if load checks, FALSE if
                                      //     TPM2_Create()
    TPMI_DH_OBJECT    parentHandle,   // IN: input parent handle
    TPMT_PUBLIC      *publicArea      // IN: public area of the object
    )
{

    // Checks for an asymmetric key
    if(CryptIsAsymAlgorithm(publicArea->type))
    {
        TPMT_ASYM_SCHEME        *keyScheme;
        keyScheme = &publicArea->parameters.asymDetail.scheme;

        // An asymmetric key can't be injected
        // This is only checked when creating an object
        if(!load && (publicArea->objectAttributes.sensitiveDataOrigin == CLEAR))
            return TPM_RC_ATTRIBUTES;

        if(load && !CryptAreKeySizesConsistent(publicArea))
            return TPM_RC_KEY;

        // Keys that are both signing and decrypting must have TPM_ALG_NULL
        // for scheme
        if(    publicArea->objectAttributes.sign == SET
           && publicArea->objectAttributes.decrypt == SET
           && keyScheme->scheme != TPM_ALG_NULL)
            return TPM_RC_SCHEME;

        // A restrict sign key must have a non-NULL scheme
        if(     publicArea->objectAttributes.restricted == SET
            &&  publicArea->objectAttributes.sign == SET
            &&  keyScheme->scheme == TPM_ALG_NULL)
            return TPM_RC_SCHEME;

        // Keys must have a valid sign or decrypt scheme, or a TPM_ALG_NULL
        // scheme
        // NOTE: The unmarshaling for a public area will unmarshal based on the
        // object type. If the type is an RSA key, then only RSA schemes will be
        // allowed because a TPMI_ALG_RSA_SCHEME will be unmarshaled and it
```

```
        // consists only of those algorithms that are allowed with an RSA key.
        // This means that there is no need to again make sure that the algorithm
        // is compatible with the object type.
        if(   keyScheme->scheme != TPM_ALG_NULL
           && (   (   publicArea->objectAttributes.sign == SET
                   && !CryptIsSignScheme(keyScheme->scheme)
                  )
               || (   publicArea->objectAttributes.decrypt == SET
                   && !CryptIsDecryptScheme(keyScheme->scheme)
                  )
              )
          )
            return TPM_RC_SCHEME;

        // Special checks for an ECC key
#ifdef TPM_ALG_ECC
        if(publicArea->type == TPM_ALG_ECC)
        {
            TPM_ECC_CURVE        curveID = publicArea->parameters.eccDetail.curveID;
            const TPMT_ECC_SCHEME  *curveScheme = CryptGetCurveSignScheme(curveID);
            // The curveId must be valid or the unmarshaling is busted.
            pAssert(curveScheme != NULL);

            // If the curveID requires a specific scheme, then the key must select
            // the same scheme
            if(curveScheme->scheme != TPM_ALG_NULL)
            {
                if(keyScheme->scheme != curveScheme->scheme)
                    return TPM_RC_SCHEME;
                // The scheme can allow any hash, or not...
                if(   curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
                   && (   keyScheme->details.anySig.hashAlg
                       != curveScheme->details.anySig.hashAlg
                      )
                  )
                    return TPM_RC_SCHEME;
            }
            // For now, the KDF must be TPM_ALG_NULL
            if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
                return TPM_RC_KDF;
        }
#endif

        // Checks for a storage key (restricted + decryption)
        if(   publicArea->objectAttributes.restricted == SET
           && publicArea->objectAttributes.decrypt == SET)
        {
            // A storage key must have a valid protection key
            if(   publicArea->parameters.asymDetail.symmetric.algorithm
               == TPM_ALG_NULL)
                return TPM_RC_SYMMETRIC;

            // A storage key must have a null scheme
            if(publicArea->parameters.asymDetail.scheme.scheme != TPM_ALG_NULL)
                return TPM_RC_SCHEME;

            // A storage key must match its parent algorithms unless
            // it is duplicable or a primary (including Temporary Primary Objects)
            if(   HandleGetType(parentHandle) != TPM_HT_PERMANENT
               && publicArea->objectAttributes.fixedParent == SET
              )
            {
                // If the object to be created is a storage key, and is fixedParent,
                // its crypto set has to match its parent's crypto set. TPM_RC_TYPE,
                // TPM_RC_HASH or TPM_RC_ASYMMETRIC may be returned at this point
                return EqualCryptSet(publicArea,
```

```
                                             &(ObjectGet(parentHandle)->publicArea));
                }
            }
            else
            {
                // Non-storage keys must have TPM_ALG_NULL for the symmetric algorithm
                if(   publicArea->parameters.asymDetail.symmetric.algorithm
                   != TPM_ALG_NULL)
                    return TPM_RC_SYMMETRIC;

            }// End of asymmetric decryption key checks
        } // End of asymmetric checks

        // Check for bit attributes
        else if(publicArea->type == TPM_ALG_KEYEDHASH)
        {
            TPMT_KEYEDHASH_SCHEME    *scheme
                = &publicArea->parameters.keyedHashDetail.scheme;
            // If both sign and decrypt are set the scheme must be TPM_ALG_NULL
            // and the scheme selected when the key is used.
            // If neither sign nor decrypt is set, the scheme must be TPM_ALG_NULL
            // because this is a data object.
            if(      publicArea->objectAttributes.sign
                ==   publicArea->objectAttributes.decrypt)
            {
                if(scheme->scheme != TPM_ALG_NULL)
                    return TPM_RC_SCHEME;
                return TPM_RC_SUCCESS;
            }
            // If this is a decryption key, make sure that is is XOR and that there
            // is a KDF
            else if(publicArea->objectAttributes.decrypt)
            {
                if(   scheme->scheme != TPM_ALG_XOR
                   || scheme->details.xor.hashAlg == TPM_ALG_NULL)
                    return TPM_RC_SCHEME;
                if(scheme->details.xor.kdf == TPM_ALG_NULL)
                    return TPM_RC_KDF;
                return TPM_RC_SUCCESS;

            }
            // only supported signing scheme for keyedHash object is HMAC
            if(   scheme->scheme != TPM_ALG_HMAC
               || scheme->details.hmac.hashAlg == TPM_ALG_NULL)
                return TPM_RC_SCHEME;

            // end of the checks for keyedHash
            return TPM_RC_SUCCESS;
        }
        else if (publicArea->type == TPM_ALG_SYMCIPHER)
        {
            // Must be a decrypting key and may not be a signing key
            if(   publicArea->objectAttributes.decrypt == CLEAR
               || publicArea->objectAttributes.sign == SET
              )
                return TPM_RC_ATTRIBUTES;
        }
        else
            return TPM_RC_TYPE;

        return TPM_RC_SUCCESS;
    }
```


8.6.3.3 PublicAttributesValidation()

This function validates the values in the public area of an object. This function is called by TPM2_Create(), TPM2_Load(), and TPM2_CreatePrimary().

Table 24

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
TPM_RC_ATTRIBUTES	*fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *publicArea*
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unexpected object type; or non-duplicable storage key and its parent have different types

```
TPM_RC
PublicAttributesValidation(
    BOOL                 load,          // IN: TRUE if load checks, FALSE if
                                        //     TPM2_Create()
    TPMI_DH_OBJECT     parentHandle,   // IN: input parent handle
    TPMT_PUBLIC       *publicArea      // IN: public area of the object
    )
{
    OBJECT              *parentObject = NULL;

    if(HandleGetType(parentHandle) != TPM_HT_PERMANENT)
        parentObject = ObjectGet(parentHandle);

    // Check authPolicy digest consistency
    if(   publicArea->authPolicy.t.size != 0
       && (   publicArea->authPolicy.t.size
           != CryptGetHashDigestSize(publicArea->nameAlg)
          )
      )
        return TPM_RC_SIZE;

    // If the parent is fixedTPM (including a Primary Object) the object must have
    // the same value for fixedTPM and fixedParent
    if(     parentObject == NULL
        || parentObject->publicArea.objectAttributes.fixedTPM == SET)
    {
        if(   publicArea->objectAttributes.fixedParent
           != publicArea->objectAttributes.fixedTPM
```

```
            )
                return TPM_RC_ATTRIBUTES;
    }
    else
        // The parent is not fixedTPM so the object can't be fixedTPM
        if(publicArea->objectAttributes.fixedTPM == SET)
            return  TPM_RC_ATTRIBUTES;

    // A restricted object cannot be both sign and decrypt and it can't be neither
    // sign nor decrypt
    if (   publicArea->objectAttributes.restricted == SET
        && (   publicArea->objectAttributes.decrypt
            == publicArea->objectAttributes.sign)
       )
        return TPM_RC_ATTRIBUTES;

    // A fixedTPM object can not have encryptedDuplication bit SET
    if(    publicArea->objectAttributes.fixedTPM == SET
        && publicArea->objectAttributes.encryptedDuplication == SET)
        return TPM_RC_ATTRIBUTES;

    // If a parent object has fixedTPM CLEAR, the child must have the
    // same encryptedDuplication value as its parent.
    // Primary objects are considered to have a fixedTPM parent (the seeds).
   if(       (   parentObject != NULL
              && parentObject->publicArea.objectAttributes.fixedTPM == CLEAR)
       // Get here if parent is not fixed TPM
       &&  (   publicArea->objectAttributes.encryptedDuplication
            != parentObject->publicArea.objectAttributes.encryptedDuplication
           )
      )
        return TPM_RC_ATTRIBUTES;

   return SchemeChecks(load, parentHandle, publicArea);
}
```

8.6.3.4 FillInCreationData()

Fill in creation data for an object.

```
void
FillInCreationData(
    TPMI_DH_OBJECT            parentHandle,  // IN: handle of parent
    TPMI_ALG_HASH             nameHashAlg,   // IN: name hash algorithm
    TPML_PCR_SELECTION       *creationPCR,   // IN: PCR selection
    TPM2B_DATA               *outsideData,   // IN: outside data
    TPM2B_CREATION_DATA      *outCreation,   // OUT: creation data for output
    TPM2B_DIGEST             *creationDigest // OUT: creation digest
    )
{
    BYTE                    creationBuffer[sizeof(TPMS_CREATION_DATA)];
    BYTE                   *buffer;
    HASH_STATE              hashState;

    // Fill in TPMS_CREATION_DATA in outCreation

    // Compute PCR digest
    PCRComputeCurrentDigest(nameHashAlg, creationPCR,
                            &outCreation->t.creationData.pcrDigest);

    // Put back PCR selection list
    outCreation->t.creationData.pcrSelect = *creationPCR;

    // Get locality
```



```
    outCreation->t.creationData.locality
        = LocalityGetAttributes(_plat__LocalityGet());

    outCreation->t.creationData.parentNameAlg = TPM_ALG_NULL;

    // If the parent is is either a primary seed or TPM_ALG_NULL, then  the Name
    // and QN of the parent are the parent's handle.
    if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
    {
        BYTE        *buffer = &outCreation->t.creationData.parentName.t.name[0];
        outCreation->t.creationData.parentName.t.size =
            TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);

        // Parent qualified name of a Temporary Object is the same as parent's
        // name
        MemoryCopy2B(&outCreation->t.creationData.parentQualifiedName.b,
                     &outCreation->t.creationData.parentName.b,
                    sizeof(outCreation->t.creationData.parentQualifiedName.t.name));

    }
    else         // Regular object
    {
        OBJECT          *parentObject = ObjectGet(parentHandle);

        // Set name algorithm
        outCreation->t.creationData.parentNameAlg =
            parentObject->publicArea.nameAlg;
        // Copy parent name
        outCreation->t.creationData.parentName = parentObject->name;

        // Copy parent qualified name
        outCreation->t.creationData.parentQualifiedName =
            parentObject->qualifiedName;
    }

    // Copy outside information
    outCreation->t.creationData.outsideInfo = *outsideData;

    // Marshal creation data to canonical form
    buffer = creationBuffer;
    outCreation->t.size = TPMS_CREATION_DATA_Marshal(&outCreation->t.creationData,
                          &buffer, NULL);

    // Compute hash for creation field in public template
    creationDigest->t.size = CryptStartHash(nameHashAlg, &hashState);
    CryptUpdateDigest(&hashState, outCreation->t.size, creationBuffer);
    CryptCompleteHash2B(&hashState, &creationDigest->b);

    return;
}
```


8.6.3.5 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed. It returns a pointer to the seed.

```
TPM2B_SEED*
GetSeedForKDF(
    TPM_HANDLE       protectorHandle,    // IN: the protector handle
    TPM2B_SEED      *seedIn              // IN: the optional input seed
    )
{
    OBJECT              *protector = NULL; // Pointer to the protector

```

```
    // Get seed for encryption key.  Use input seed if provided.
    // Otherwise, using protector object's seedValue.  TPM_RH_NULL is the only
    // exception that we may not have a loaded object as protector.  In such a
    // case, use nullProof as seed.
    if(seedIn != NULL)
    {
        return seedIn;
    }
    else
    {
        if(protectorHandle == TPM_RH_NULL)
        {
            return (TPM2B_SEED *) &gr.nullProof;
        }
        else
        {
            protector = ObjectGet(protectorHandle);
            return (TPM2B_SEED *) &protector->sensitive.seedValue;
        }
    }
}
```

8.6.3.6 ProduceOuterWrap()

This function produce outer wrap for a buffer containing the sensitive data. It requires the sensitive data being marshaled to the *outerBuffer*, with the leading bytes reserved for integrity hash. If iv is used, iv space should be reserved at the beginning of the buffer. It assumes the sensitive data starts at address (*outerBuffer* + integrity size {+ iv size}). This function performs:

a) Add IV before sensitive area if required

b) encrypt sensitive data, if iv is required, encrypt by iv. otherwise, encrypted by a NULL iv

c) add HMAC integrity at the beginning of the buffer It returns the total size of blob with outer wrap.

```
UINT16
ProduceOuterWrap(
    TPM_HANDLE       protector,    // IN: The handle of the object that provides
                                   //     protection.  For object, it is parent
                                   //     handle. For credential, it is the handle
                                   //     of encrypt object.
    TPM2B_NAME      *name,         // IN: the name of the object
    TPM_ALG_ID       hashAlg,      // IN: hash algorithm for outer wrap
    TPM2B_SEED      *seed,         // IN: an external seed may be provided for
                                   //     duplication blob. For non duplication
                                   //     blob, this parameter should be NULL
    BOOL             useIV,        // IN: indicate if an IV is used
    UINT16           dataSize,     // IN: the size of sensitive data, excluding the
                                   //     leading integrity buffer size or the
                                   //     optional iv size
    BYTE            *outerBuffer   // IN/OUT: outer buffer with sensitive data in
                                   //     it
    )
{
    TPM_ALG_ID      symAlg;
    UINT16          keyBits;
    TPM2B_SYM_KEY   symKey;
    TPM2B_IV        ivRNG;         // IV from RNG
    TPM2B_IV        *iv = NULL;
    UINT16          ivSize = 0;    // size of iv area, including the size field

    BYTE            *sensitiveData; // pointer to the sensitive data

    TPM2B_DIGEST    integrity;
    UINT16          integritySize;
```



```
    BYTE            *buffer;        // Auxiliary buffer pointer

    // Compute the beginning of sensitive data.  The outer integrity should
    // always exist if this function function is called to make an outer wrap
    integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);
    sensitiveData = outerBuffer + integritySize;

    // If iv is used, adjust the pointer of sensitive data and add iv before it
    if(useIV)
    {
        ivSize = GetIV2BSize(protector);

        // Generate IV from RNG.  The iv data size should be the total IV area
        // size minus the size of size field
        ivRNG.t.size = ivSize - sizeof(UINT16);
        CryptGenerateRandom(ivRNG.t.size, ivRNG.t.buffer);

        // Marshal IV to buffer
        buffer = sensitiveData;
        TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);

        // adjust sensitive data starting after IV area
        sensitiveData += ivSize;

        // Use iv for encryption
        iv = &ivRNG;
    }

    // Compute symmetric key parameters for outer buffer encryption
    ComputeProtectionKeyParms(protector, hashAlg, name, seed,
                              &symAlg, &keyBits, &symKey);
    // Encrypt inner buffer in place
    CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
                          TPM_ALG_CFB, symKey.t.buffer, iv, dataSize,
                          sensitiveData);

    // Compute outer integrity.  Integrity computation includes the optional IV
    // area
    ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
                          outerBuffer + integritySize, &integrity);

    // Add integrity at the beginning of outer buffer
    buffer = outerBuffer;
    TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);

    // return the total size in outer wrap
    return dataSize + integritySize + ivSize;

}
```

8.6.3.7 UnwrapOuter()

This function remove the outer wrap of a blob containing sensitive data. This function performs:

a) check integrity of outer blob

b) decrypt outer blob

Table 25

Error Returns	Meaning
TPM_RC_INSUFFICIENT	error during sensitive data unmarshaling
TPM_RC_INTEGRITY	sensitive data integrity is broken
TPM_RC_SIZE	error during sensitive data unmarshaling
TPM_RC_VALUE	IV size for CFB does not match the encryption algorithm block size

```
TPM_RC
UnwrapOuter(
    TPM_HANDLE       protector,     // IN: The handle of the object that provides
                                    //     protection.  For object, it is parent
                                    //     handle. For credential, it is the handle
                                    //     of encrypt object.
    TPM2B_NAME      *name,          // IN: the name of the object
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for outer wrap
    TPM2B_SEED      *seed,          // IN: an external seed may be provided for
                                    //     duplication blob. For non duplication
                                    //     blob, this parameter should be NULL.
    BOOL             useIV,         // IN: indicates if an IV is used
    UINT16           dataSize,      // IN: size of sensitive data in outerBuffer,
                                    //     including the leading integrity buffer
                                    //     size, and an optional iv area
    BYTE            *outerBuffer    // IN/OUT: sensitive data
    )
{
    TPM_RC          result;
    TPM_ALG_ID      symAlg = TPM_ALG_NULL;
    TPM2B_SYM_KEY   symKey;
    UINT16          keyBits = 0;
    TPM2B_IV        ivIn;               // input IV retrieved from input buffer
    TPM2B_IV        *iv = NULL;

    BYTE            *sensitiveData;     // pointer to the sensitive data

    TPM2B_DIGEST    integrityToCompare;
    TPM2B_DIGEST    integrity;
    INT32           size;

    // Unmarshal integrity
    sensitiveData = outerBuffer;
    size = (INT32) dataSize;
    result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
    if(result == TPM_RC_SUCCESS)
    {
        // Compute integrity to compare
        ComputeOuterIntegrity(name, protector, hashAlg, seed,
                              (UINT16) size, sensitiveData,
                              &integrityToCompare);

        // Compare outer blob integrity
        if(!Memory2BEqual(&integrity.b, &integrityToCompare.b))
            return TPM_RC_INTEGRITY;

        // Get the symmetric algorithm parameters used for encryption
        ComputeProtectionKeyParms(protector, hashAlg, name, seed,
                                  &symAlg, &keyBits, &symKey);

        // Retrieve IV if it is used
        if(useIV)
        {
            result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
```



```
            if(result == TPM_RC_SUCCESS)
            {
                // The input iv size for CFB must match the encryption algorithm
                // block size
                if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
                    result = TPM_RC_VALUE;
                else
                    iv = &ivIn;
            }
        }
    }
    // If no errors, decrypt private in place
    if(result == TPM_RC_SUCCESS)
        CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
                              TPM_ALG_CFB, symKey.t.buffer, iv,
                              (UINT16) size, sensitiveData);

    return result;

}
```

8.6.3.8 SensitiveToPrivate()

This function prepare the private blob for off the chip storage The operations in this function:

a) marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply encryption to the sensitive area.

c) apply outer integrity computation.

```
void
SensitiveToPrivate(
    TPMT_SENSITIVE  *sensitive,     // IN: sensitive structure
    TPM2B_NAME      *name,          // IN: the name of the object
    TPM_HANDLE       parentHandle,  // IN: The parent's handle
    TPM_ALG_ID       nameAlg,       // IN: hash algorithm in public area.  This
                                    //     parameter is used when parentHandle is
                                    //     NULL, in which case the object is
                                    //     temporary.
    TPM2B_PRIVATE   *outPrivate     // OUT: output private structure
    )
{
    BYTE                *buffer;                // Auxiliary buffer pointer
    BYTE                *sensitiveData;         // pointer to the sensitive data
    UINT16              dataSize;               // data blob size
    TPMI_ALG_HASH       hashAlg;                // hash algorithm for integrity
    UINT16              integritySize;
    UINT16              ivSize;

    pAssert(name != NULL && name->t.size != 0);

    // Find the hash algorithm for integrity computation
    if(parentHandle == TPM_RH_NULL)
    {
        // For Temporary Object, using self name algorithm
        hashAlg = nameAlg;
    }
    else
    {
        // Otherwise, using parent's name algorithm
        hashAlg = ObjectGetNameAlg(parentHandle);
    }

    // Starting of sensitive data without wrappers
```

```
    sensitiveData = outPrivate->t.buffer;

    // Compute the integrity size
    integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);

    // Reserve space for integrity
    sensitiveData += integritySize;

    // Get iv size
    ivSize = GetIV2BSize(parentHandle);

    // Reserve space for iv
    sensitiveData += ivSize;

    // Marshal sensitive area, leaving the leading 2 bytes for size
    buffer = sensitiveData + sizeof(UINT16);
    dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);

    // Adding size before the data area
    buffer = sensitiveData;
    UINT16_Marshal(&dataSize, &buffer, NULL);

    // Adjust the dataSize to include the size field
    dataSize += sizeof(UINT16);

    // Adjust the pointer to inner buffer including the iv
    sensitiveData = outPrivate->t.buffer + ivSize;

    //Produce outer wrap, including encryption and HMAC
    outPrivate->t.size = ProduceOuterWrap(parentHandle, name, hashAlg, NULL,
                                          TRUE, dataSize, outPrivate->t.buffer);

    return;
}
```


8.6.3.9 PrivateToSensitive()

Unwrap a input private area. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

a) check the integrity HMAC of the input private area

b) decrypt the private buffer

c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Table 26

Error Returns	**Meaning**
TPM_RC_INTEGRITY	if the private area integrity is bad
TPM_RC_SENSITIVE	unmarshal errors while unmarshaling TPMS_ENCRYPT from input private
TPM_RC_VALUE	outer wrapper does not have an *iV* of the correct size

```
TPM_RC
PrivateToSensitive(
    TPM2B_PRIVATE    *inPrivate,      // IN: input private structure
    TPM2B_NAME       *name,           // IN: the name of the object
    TPM_HANDLE        parentHandle,   // IN: The parent's handle
    TPM_ALG_ID        nameAlg,        // IN: hash algorithm in public area.  It is
                                      //     passed separately because we only pass
                                      //     name, rather than the whole public area
```



```
                                    //     of the object.  This parameter is used in
                                    //     the following two cases: 1. primary
                                    //     objects. 2. duplication blob with inner
                                    //     wrap.  In other cases, this parameter
                                    //     will be ignored
    TPMT_SENSITIVE  *sensitive      // OUT: sensitive structure
    )
{
    TPM_RC          result;

    BYTE            *buffer;
    INT32           size;
    BYTE            *sensitiveData; // pointer to the sensitive data
    UINT16          dataSize;
    UINT16          dataSizeInput;
    TPMI_ALG_HASH   hashAlg;        // hash algorithm for integrity
    OBJECT          *parent = NULL;

    UINT16          integritySize;
    UINT16          ivSize;

    // Make sure that name is provided
    pAssert(name != NULL && name->t.size != 0);

    // Find the hash algorithm for integrity computation
    if(parentHandle == TPM_RH_NULL)
    {
        // For Temporary Object, using self name algorithm
        hashAlg = nameAlg;
    }
    else
    {
        // Otherwise, using parent's name algorithm
        hashAlg = ObjectGetNameAlg(parentHandle);
    }

    // unwrap outer
    result = UnwrapOuter(parentHandle, name, hashAlg, NULL, TRUE,
                         inPrivate->t.size, inPrivate->t.buffer);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Compute the inner integrity size.
    integritySize = sizeof(UINT16) + CryptGetHashDigestSize(hashAlg);

    // Get iv size
    ivSize = GetIV2BSize(parentHandle);

    // The starting of sensitive data and data size without outer wrapper
    sensitiveData = inPrivate->t.buffer + integritySize + ivSize;
    dataSize = inPrivate->t.size - integritySize - ivSize;

    // Unmarshal input data size
    buffer = sensitiveData;
    size = (INT32) dataSize;
    result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
    if(result == TPM_RC_SUCCESS)
    {
        if((dataSizeInput + sizeof(UINT16)) != dataSize)
            result = TPM_RC_SENSITIVE;
        else
        {
            // Unmarshal sensitive buffer to sensitive structure
            result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
            if(result != TPM_RC_SUCCESS || size != 0)
            {
```

```
                pAssert(   (parent == NULL)
                        || parent->publicArea.objectAttributes.fixedTPM == CLEAR);
                result = TPM_RC_SENSITIVE;
            }
            else
            {
                // Always remove trailing zeros at load so that it is not necessary
                // to check
                // each time auth is checked.
                MemoryRemoveTrailingZeros(&(sensitive->authValue));
            }
        }
    }
    return result;
}
```

8.6.3.10 SensitiveToDuplicate()

This function prepare the duplication blob from the sensitive area. The operations in this function:

a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

b) apply inner wrap to the sensitive area if required

c) apply outer wrap if required

```
void
SensitiveToDuplicate(
    TPMT_SENSITIVE          *sensitive,      // IN: sensitive structure
    TPM2B_NAME              *name,           // IN: the name of the object
    TPM_HANDLE               parentHandle,   // IN: The new parent's handle
    TPM_ALG_ID               nameAlg,        // IN: hash algorithm in public area.
                                             //     It is passed separately because
                                             //     we only pass name, rather than
                                             //     the whole public area of the
                                             //     object.
    TPM2B_SEED              *seed,           // IN: the external seed. If external
                                             //     seed is provided with size of 0,
                                             //     no outer wrap should be applied
                                             //     to duplication blob.
    TPMT_SYM_DEF_OBJECT     *symDef,         // IN: Symmetric key definition. If the
                                             //     symmetric key algorithm is NULL,
                                             //     no inner wrap should be applied.
    TPM2B_DATA              *innerSymKey,    // IN/OUT: a symmetric key may be
                                             //     provided to encrypt the inner
                                             //     wrap of a duplication blob. May
                                             //     be generated here if needed.
    TPM2B_PRIVATE           *outPrivate      // OUT: output private structure
    )
{
    BYTE            *buffer;        // Auxiliary buffer pointer
    BYTE            *sensitiveData; // pointer to the sensitive data
    TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;// The hash algorithm for outer wrap
    TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;// The hash algorithm for inner wrap
    UINT16          dataSize;       // data blob size
    BOOL            doInnerWrap = FALSE;
    BOOL            doOuterWrap = FALSE;

    // Make sure that name is provided
    pAssert(name != NULL && name->t.size != 0);

    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);

    // Starting of sensitive data without wrappers
```



```
    sensitiveData = outPrivate->t.buffer;

    // Find out if inner wrap is required
    if(symDef->algorithm != TPM_ALG_NULL)
    {
        doInnerWrap = TRUE;
        // Use self nameAlg as inner hash algorithm
        innerHash = nameAlg;
        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
    }

    // Find out if outer wrap is required
    if(seed->t.size != 0)
    {
        doOuterWrap = TRUE;
        // Use parent nameAlg as outer hash algorithm
        outerHash = ObjectGetNameAlg(parentHandle);
        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
    }

    // Marshal sensitive area, leaving the leading 2 bytes for size
    buffer = sensitiveData + sizeof(UINT16);
    dataSize = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);

    // Adding size before the data area
    buffer = sensitiveData;
    UINT16_Marshal(&dataSize, &buffer, NULL);

    // Adjust the dataSize to include the size field
    dataSize += sizeof(UINT16);

    // Apply inner wrap for duplication blob.  It includes both integrity and
    // encryption
    if(doInnerWrap)
    {
        BYTE            *innerBuffer = NULL;
        BOOL            symKeyInput = TRUE;
        innerBuffer = outPrivate->t.buffer;
        // Skip outer integrity space
        if(doOuterWrap)
            innerBuffer += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
        dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
                                         innerBuffer);

        // Generate inner encryption key if needed
        if(innerSymKey->t.size == 0)
        {
            innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
            CryptGenerateRandom(innerSymKey->t.size, innerSymKey->t.buffer);

            // TPM generates symmetric encryption.  Set the flag to FALSE
            symKeyInput = FALSE;
        }
        else
        {
            // assume the input key size should matches the symmetric definition
            pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);

        }

        // Encrypt inner buffer in place
        CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
                              symDef->keyBits.sym, TPM_ALG_CFB,
                              innerSymKey->t.buffer, NULL, dataSize,
```

```
                                  innerBuffer);

        // If the symmetric encryption key is imported, clear the buffer for
        // output
        if(symKeyInput)
            innerSymKey->t.size = 0;
    }

    // Apply outer wrap for duplication blob.  It includes both integrity and
    // encryption
    if(doOuterWrap)
    {
        dataSize = ProduceOuterWrap(parentHandle, name, outerHash, seed, FALSE,
                                    dataSize, outPrivate->t.buffer);
    }

    // Data size for output
    outPrivate->t.size = dataSize;

    return;
}
```

8.6.3.11 DuplicateToSensitive()

Unwrap a duplication blob. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

a) check the integrity HMAC of the input private area

b) decrypt the private buffer

c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Table 27

Error Returns	Meaning
TPM_RC_INSUFFICIENT	unmarshaling sensitive data from *inPrivate* failed
TPM_RC_INTEGRITY	*inPrivate* data integrity is broken
TPM_RC_SIZE	unmarshaling sensitive data from *inPrivate* failed

```
TPM_RC
DuplicateToSensitive(
    TPM2B_PRIVATE           *inPrivate,      // IN: input private structure
    TPM2B_NAME              *name,           // IN: the name of the object
    TPM_HANDLE               parentHandle,   // IN: The parent's handle
    TPM_ALG_ID               nameAlg,        // IN: hash algorithm in public area.
    TPM2B_SEED              *seed,           // IN: an external seed may be provided.
                                             //     If external seed is provided with
                                             //     size of 0, no outer wrap is
                                             //     applied
    TPMT_SYM_DEF_OBJECT     *symDef,         // IN: Symmetric key definition. If the
                                             //     symmetric key algorithm is NULL,
                                             //     no inner wrap is applied
    TPM2B_DATA              *innerSymKey,    // IN: a symmetric key may be provided
                                             //     to decrypt the inner wrap of a
                                             //     duplication blob.
    TPMT_SENSITIVE          *sensitive       // OUT: sensitive structure
    )
{
    TPM_RC          result;

    BYTE            *buffer;
```



```
    INT32           size;
    BYTE            *sensitiveData; // pointer to the sensitive data
    UINT16          dataSize;
    UINT16          dataSizeInput;

    // Make sure that name is provided
    pAssert(name != NULL && name->t.size != 0);

    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);

    // Starting of sensitive data
    sensitiveData = inPrivate->t.buffer;
    dataSize = inPrivate->t.size;

    // Find out if outer wrap is applied
    if(seed->t.size != 0)
    {
        TPMI_ALG_HASH   outerHash = TPM_ALG_NULL;

        // Use parent nameAlg as outer hash algorithm
        outerHash = ObjectGetNameAlg(parentHandle);
        result = UnwrapOuter(parentHandle, name, outerHash, seed, FALSE,
                             dataSize, sensitiveData);
        if(result != TPM_RC_SUCCESS)
            return result;

        // Adjust sensitive data pointer and size
        sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
        dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
    }
    // Find out if inner wrap is applied
    if(symDef->algorithm != TPM_ALG_NULL)
    {
        TPMI_ALG_HASH   innerHash = TPM_ALG_NULL;

        // assume the input key size should matches the symmetric definition
        pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);

        // Decrypt inner buffer in place
        CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
                              symDef->keyBits.sym, TPM_ALG_CFB,
                              innerSymKey->t.buffer, NULL, dataSize,
                              sensitiveData);

        // Use self nameAlg as inner hash algorithm
        innerHash = nameAlg;

        // Check inner integrity
        result = CheckInnerIntegrity(name, innerHash, dataSize, sensitiveData);
        if(result != TPM_RC_SUCCESS)
            return result;

        // Adjust sensitive data pointer and size
        sensitiveData += sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
        dataSize -= sizeof(UINT16) + CryptGetHashDigestSize(innerHash);
    }

    // Unmarshal input data size
    buffer = sensitiveData;
    size = (INT32) dataSize;
    result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
    if(result == TPM_RC_SUCCESS)
    {
        if((dataSizeInput + sizeof(UINT16)) != dataSize)
            result = TPM_RC_SIZE;
```

```
        else
        {
            // Unmarshal sensitive buffer to sensitive structure
            result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
            // if the results is OK make sure that all the data was unmarshaled
            if(result == TPM_RC_SUCCESS && size != 0)
                result = TPM_RC_SIZE;
        }
    }
    // Always remove trailing zeros at load so that it is not necessary to check
    // each time auth is checked.
    if(result == TPM_RC_SUCCESS)
        MemoryRemoveTrailingZeros(&(sensitive->authValue));
    return result;
}
```

8.6.3.12 SecretToCredential()

This function prepare the credential blob from a secret (a TPM2B_DIGEST) The operations in this function:

a) marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT

b) encrypt the private buffer, excluding the leading integrity HMAC area

c) compute integrity HMAC and append to the beginning of the buffer.

d) Set the total size of TPM2B_ID_OBJECT buffer

```
void
SecretToCredential(
    TPM2B_DIGEST        *secret,        // IN: secret information
    TPM2B_NAME          *name,          // IN: the name of the object
    TPM2B_SEED          *seed,          // IN: an external seed.
    TPM_HANDLE           protector,     // IN: The protector's handle
    TPM2B_ID_OBJECT     *outIDObject    // OUT: output credential
    )
{
    BYTE                *buffer;        // Auxiliary buffer pointer
    BYTE                *sensitiveData; // pointer to the sensitive data
    TPMI_ALG_HASH        outerHash;     // The hash algorithm for outer wrap
    UINT16               dataSize;      // data blob size

    pAssert(secret != NULL && outIDObject != NULL);

    // use protector's name algorithm as outer hash
    outerHash = ObjectGetNameAlg(protector);

    // Marshal secret area to credential buffer, leave space for integrity
    sensitiveData = outIDObject->t.credential
                    + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);

    // Marshal secret area
    buffer = sensitiveData;
    dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);

    // Apply outer wrap
    outIDObject->t.size = ProduceOuterWrap(protector,
                                           name,
                                           outerHash,
                                           seed,
                                           FALSE,
                                           dataSize,
                                           outIDObject->t.credential);
    return;
```



```
}
```

8.6.3.13 CredentialToSecret()

Unwrap a credential. Check the integrity, decrypt and retrieve data to a TPM2B_DIGEST structure. The operations in this function:

a) check the integrity HMAC of the input credential area

b) decrypt the credential buffer

c) unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

Table 28

Error Returns	Meaning
TPM_RC_INSUFFICIENT	error during credential unmarshaling
TPM_RC_INTEGRITY	credential integrity is broken
TPM_RC_SIZE	error during credential unmarshaling
TPM_RC_VALUE	IV size does not match the encryption algorithm block size


```
TPM_RC
CredentialToSecret(
    TPM2B_ID_OBJECT     *inIDObject,    // IN: input credential blob
    TPM2B_NAME          *name,          // IN: the name of the object
    TPM2B_SEED          *seed,          // IN: an external seed.
    TPM_HANDLE           protector,     // IN: The protector's handle
    TPM2B_DIGEST        *secret         // OUT: secret information
    )
{
    TPM_RC                   result;
    BYTE                    *buffer;
    INT32                    size;
    TPMI_ALG_HASH            outerHash;     // The hash algorithm for outer wrap
    BYTE                    *sensitiveData; // pointer to the sensitive data
    UINT16                   dataSize;

    // use protector's name algorithm as outer hash
    outerHash = ObjectGetNameAlg(protector);

    // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
    result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
                         inIDObject->t.size, inIDObject->t.credential);
    if(result == TPM_RC_SUCCESS)
    {
        // Compute the beginning of sensitive data
        sensitiveData = inIDObject->t.credential
                        + sizeof(UINT16) + CryptGetHashDigestSize(outerHash);
        dataSize = inIDObject->t.size
                   - (sizeof(UINT16) + CryptGetHashDigestSize(outerHash));

        // Unmarshal secret buffer to TPM2B_DIGEST structure
        buffer = sensitiveData;
        size = (INT32) dataSize;
        result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
        // If there were no other unmarshaling errors, make sure that the
        // expected amount of data was recovered
        if(result == TPM_RC_SUCCESS && size != 0)
            return TPM_RC_SIZE;
    }
    return result;
```

```
}
```

9 Subsystem

9.1 CommandAudit.c

9.1.1 Introduction

This file contains the functions that support command audit.

9.1.2 Includes

```
#include "InternalRoutines.h"
```

9.1.3 Functions

9.1.3.1 CommandAuditPreInstall_Init()

This function initializes the command audit list. This function is simulates the behavior of manufacturing. A function is used instead of a structure definition because this is easier than figuring out the initialization value for a bit array.

This function would not be implemented outside of a manufacturing or simulation environment.

```
void
CommandAuditPreInstall_Init(
    void
    )
{
    // Clear all the audit commands
    MemorySet(gp.auditComands, 0x00,
              ((TPM_CC_LAST - TPM_CC_FIRST + 1) + 7) / 8);

    // TPM_CC_SetCommandCodeAuditStatus always being audited
    if(CommandIsImplemented(TPM_CC_SetCommandCodeAuditStatus))
        CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);

    // Set initial command audit hash algorithm to be context integrity hash
    // algorithm
    gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;

    // Set up audit counter to be 0
    gp.auditCounter = 0;

    // Write command audit persistent data to NV
    NvWriteReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
    NvWriteReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
    NvWriteReserved(NV_AUDIT_COUNTER, &gp.auditCounter);

    return;
}
```

9.1.3.2 CommandAuditStartup()

This function clears the command audit digest on a TPM Reset.

```
void
CommandAuditStartup(
    STARTUP_TYPE     type             // IN: start up type
    )
```

```
{
    if(type == SU_RESET)
    {
        // Reset the digest size to initialize the digest
        gr.commandAuditDigest.t.size = 0;
    }

}
```

9.1.3.3 CommandAuditSet()

This function will SET the audit flag for a command. This function will not SET the audit flag for a command that is not implemented. This ensures that the audit status is not SET when TPM2_GetCapability() is used to read the list of audited commands.

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Table 29

Return Value	Meaning
TRUE	the command code audit status was changed
FALSE	the command code audit status was not changed

```
BOOL
CommandAuditSet(
    TPM_CC          commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Only SET a bit if the corresponding command is implemented
    if(CommandIsImplemented(commandCode))
    {
        // Can't audit shutdown
        if(commandCode != TPM_CC_Shutdown)
        {
            bitPos = commandCode - TPM_CC_FIRST;
            if(!BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
            {
                // Set bit
                BitSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}
```

9.1.3.4 CommandAuditClear()

This function will CLEAR the audit flag for a command. It will not CLEAR the audit flag for TPM_CC_SetCommandCodeAuditStatus().

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Table 30

Return Value	Meaning
TRUE	the command code audit status was changed
FALSE	the command code audit status was not changed

```
BOOL
CommandAuditClear(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Do nothing if the command is not implemented
    if(CommandIsImplemented(commandCode))
    {
        // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
        // cleared
        if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
        {
            bitPos = commandCode - TPM_CC_FIRST;
            if(BitIsSet(bitPos, &gp.auditComands[0], sizeof(gp.auditComands)))
            {
                // Clear bit
                BitClear(bitPos, &gp.auditComands[0], sizeof(gp.auditComands));
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}
```

9.1.3.5 CommandAuditIsRequired()

This function indicates if the audit flag is SET for a command.

Table 31

Return Value	Meaning
TRUE	if command is audited
FALSE	if command is not audited

```
BOOL
CommandAuditIsRequired(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    bitPos = commandCode - TPM_CC_FIRST;

    // Check the bit map.  If the bit is SET, command audit is required
    if((gp.auditComands[bitPos/8] & (1 << (bitPos % 8))) != 0)
        return TRUE;
    else
        return FALSE;

}
```

9.1.3.6 CommandAuditCapGetCCList()

This function returns a list of commands that have their audit bit SET.

The list starts at the input *commandCode*.

Table 32

Return Value	Meaning
YES	if there are more command code available
NO	all the available command code has been returned

```
TPMI_YES_NO
CommandAuditCapGetCCList(
    TPM_CC           commandCode,    // IN: start command code
    UINT32           count,          // IN: count of returned TPM_CC
    TPML_CC         *commandList     // OUT: list of TPM_CC
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // Initialize output handle list
    commandList->count = 0;

    // The maximum count of command we may return is MAX_CAP_CC
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

    // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
    if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;

    // Collect audit commands
    for(i = commandCode; i <= TPM_CC_LAST; i++)
    {
        if(CommandAuditIsRequired(i))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to it
                commandList->commandCodes[commandList->count] = i;
                commandList->count++;
            }
            else
            {
                // If the return list is full but we still have command
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }

    return more;

}
```

9.1.3.7 CommandAuditGetDigest

This command is used to create a digest of the commands being audited. The commands are processed in ascending numeric order with a list of TPM_CC being added to a hash. This operates as if all the audited command codes were concatenated and then hashed.

```
void
CommandAuditGetDigest(
    TPM2B_DIGEST    *digest         // OUT: command digest
    )
{
    TPM_CC                      i;
    HASH_STATE                  hashState;

    // Start hash
    digest->t.size = CryptStartHash(gp.auditHashAlg, &hashState);

    // Add command code
    for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
    {
        if(CommandAuditIsRequired(i))
        {
            CryptUpdateDigestInt(&hashState, sizeof(i), &i);
        }
    }

    // Complete hash
    CryptCompleteHash2B(&hashState, &digest->b);

    return;
}
```

9.2 DA.c

9.2.1 Introduction

This file contains the functions and data definitions relating to the dictionary attack logic.

9.2.2 Includes and Data Definitions

```
#define DA_C
#include "InternalRoutines.h"
```

9.2.3 Functions

9.2.3.1 DAPreInstall_Init()

This function initializes the DA parameters to their manufacturer-default values. The default values are determined by a platform-specific specification.

This function should not be called outside of a manufacturing or simulation environment.

The DA parameters will be restored to these initial values by TPM2_Clear().

```
void
DAPreInstall_Init(
    void
    )
{
```

```
    gp.failedTries = 0;
    gp.maxTries = 3;
    gp.recoveryTime = 1000;          // in seconds (~16.67 minutes)
    gp.lockoutRecovery = 1000;       // in seconds
    gp.lockOutAuthEnabled = TRUE;    // Use of lockoutAuth is enabled

    // Record persistent DA parameter changes to NV
    NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvWriteReserved(NV_MAX_TRIES, &gp.maxTries);
    NvWriteReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvWriteReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);

    return;
}
```

9.2.3.2 DAStartup()

This function is called by TPM2_Startup() to initialize the DA parameters. In the case of Startup(CLEAR), use of *lockoutAuth* will be enabled if the lockout recovery time is 0. Otherwise, *lockoutAuth* will not be enabled until the TPM has been continuously powered for the *lockoutRecovery* time.

This function requires that NV be available and not rate limiting.

```
void
DAStartup(
    STARTUP_TYPE     type            // IN: startup type
    )
{
    // For TPM Reset, if lockoutRecovery is 0, enable use of lockoutAuth.
    if(type == SU_RESET)
    {
        if(gp.lockoutRecovery == 0)
        {
            gp.lockOutAuthEnabled = TRUE;
            // Record the changes to NV
            NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
        }
    }

    // If DA has not been disabled and the previous shutdown is not orderly
    // failedTries is not already at its maximum then increment 'failedTries'
    if(    gp.recoveryTime != 0
        && g_prevOrderlyState == SHUTDOWN_NONE
        && gp.failedTries < gp.maxTries)
    {
        gp.failedTries++;
        // Record the change to NV
        NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
    }

    // Reset self healing timers
    s_selfHealTimer = g_time;
    s_lockoutTimer = g_time;

    return;
}
```

9.2.3.3 DARegisterFailure()

This function is called when a authorization failure occurs on an entity that is subject to dictionary-attack protection. When a DA failure is triggered, register the failure by resetting the relevant self-healing timer to the current time.


```
void
DARegisterFailure(
    TPM_HANDLE       handle          // IN: handle for failure
    )
{
    // Reset the timer associated with lockout if the handle is the lockout auth.
    if(handle == TPM_RH_LOCKOUT)
        s_lockoutTimer = g_time;
    else
        s_selfHealTimer = g_time;

    return;
}
```

9.2.3.4 DASelfHeal()

This function is called to check if sufficient time has passed to allow decrement of *failedTries* or to re-enable use of *lockoutAuth*.

This function should be called when the time interval is updated.

```
void
DASelfHeal(
    void
    )
{
    // Regular auth self healing logic
    // If no failed authorization tries, do nothing. Otherwise, try to
    // decrease failedTries
    if(gp.failedTries != 0)
    {
        // if recovery time is 0, DA logic has been disabled.  Clear failed tries
        // immediately
        if(gp.recoveryTime == 0)
        {
            gp.failedTries = 0;
            // Update NV record
            NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
        }
        else
        {
            UINT64          decreaseCount;

            // In the unlikely event that failedTries should become larger than
            // maxTries
            if(gp.failedTries > gp.maxTries)
                gp.failedTries = gp.maxTries;

            // How much can failedTried be decreased
            decreaseCount = ((g_time - s_selfHealTimer) / 1000) / gp.recoveryTime;

            if(gp.failedTries <= (UINT32) decreaseCount)
                // should not set failedTries below zero
                gp.failedTries = 0;
            else
                gp.failedTries -= (UINT32) decreaseCount;

            // the cast prevents overflow of the product
            s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
            if(decreaseCount != 0)
                // If there was a change to the failedTries, record the changes
                // to NV
                NvWriteReserved(NV_FAILED_TRIES, &gp.failedTries);
        }
```

```
    }

    // LockoutAuth self healing logic
    // If lockoutAuth is enabled, do nothing.  Otherwise, try to see if we
    // may enable it
    if(!gp.lockOutAuthEnabled)
    {
        // if lockout authorization recovery time is 0, a reboot is required to
        // re-enable use of lockout authorization.  Self-healing would not
        // apply in this case.
        if(gp.lockoutRecovery != 0)
        {
            if(((g_time - s_lockoutTimer)/1000) >= gp.lockoutRecovery)
            {
                gp.lockOutAuthEnabled = TRUE;
                // Record the changes to NV
                NvWriteReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);
            }
        }
    }

    return;
}
```

9.3 Hierarchy.c

9.3.1 Introduction

This file contains the functions used for managing and accessing the hierarchy-related values.

9.3.2 Includes

```
#include "InternalRoutines.h"
```

9.3.3 Functions

9.3.3.1 HierarchyPreInstall()

This function performs the initialization functions for the hierarchy when the TPM is simulated. This function should not be called if the TPM is not in a manufacturing mode at the manufacturer, or in a simulated environment.

```
void
HierarchyPreInstall_Init(
    void
    )
{
    // Allow lockout clear command
    gp.disableClear = FALSE;

    // Initialize Primary Seeds
    gp.EPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
    gp.SPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
    gp.PPSeed.t.size = PRIMARY_SEED_SIZE;
    CryptGenerateRandom(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);

    // Initialize owner, endorsement and lockout auth
```



```
    gp.ownerAuth.t.size = 0;
    gp.endorsementAuth.t.size = 0;
    gp.lockoutAuth.t.size = 0;

    // Initialize owner, endorsement, and lockout policy
    gp.ownerAlg = TPM_ALG_NULL;
    gp.ownerPolicy.t.size = 0;
    gp.endorsementAlg = TPM_ALG_NULL;
    gp.endorsementPolicy.t.size = 0;
    gp.lockoutAlg = TPM_ALG_NULL;
    gp.lockoutPolicy.t.size = 0;

    // Initialize ehProof, shProof and phProof
    gp.phProof.t.size = PROOF_SIZE;
    gp.shProof.t.size = PROOF_SIZE;
    gp.ehProof.t.size = PROOF_SIZE;
    CryptGenerateRandom(gp.phProof.t.size, gp.phProof.t.buffer);
    CryptGenerateRandom(gp.shProof.t.size, gp.shProof.t.buffer);
    CryptGenerateRandom(gp.ehProof.t.size, gp.ehProof.t.buffer);

    // Write hierarchy data to NV
    NvWriteReserved(NV_DISABLE_CLEAR, &gp.disableClear);
    NvWriteReserved(NV_EP_SEED, &gp.EPSeed);
    NvWriteReserved(NV_SP_SEED, &gp.SPSeed);
    NvWriteReserved(NV_PP_SEED, &gp.PPSeed);
    NvWriteReserved(NV_OWNER_AUTH, &gp.ownerAuth);
    NvWriteReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
    NvWriteReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
    NvWriteReserved(NV_OWNER_ALG, &gp.ownerAlg);
    NvWriteReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
    NvWriteReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
    NvWriteReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
    NvWriteReserved(NV_LOCKOUT_ALG, &gp.lockoutAlg);
    NvWriteReserved(NV_LOCKOUT_POLICY, &gp.lockoutPolicy);
    NvWriteReserved(NV_PH_PROOF, &gp.phProof);
    NvWriteReserved(NV_SH_PROOF, &gp.shProof);
    NvWriteReserved(NV_EH_PROOF, &gp.ehProof);

    return;
}
```

9.3.3.2 HierarchyStartup()

This function is called at TPM2_Startup() to initialize the hierarchy related values.

```
void
HierarchyStartup(
    STARTUP_TYPE      type            // IN: start up type
    )
{
    // phEnable is SET on any startup
    g_phEnable = TRUE;

    // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
    // TPM_RESTART
    if(type != SU_RESUME)
    {
        gc.platformAuth.t.size = 0;
        gc.platformPolicy.t.size = 0;

        // enable the storage and endorsement hierarchies and the platformNV
        gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
    }

```

```
    // nullProof and nullSeed are updated at every TPM_RESET
    if(type == SU_RESET)
    {
        gr.nullProof.t.size = PROOF_SIZE;
        CryptGenerateRandom(gr.nullProof.t.size,
                            gr.nullProof.t.buffer);
        gr.nullSeed.t.size = PRIMARY_SEED_SIZE;
        CryptGenerateRandom(PRIMARY_SEED_SIZE, gr.nullSeed.t.buffer);
    }

    return;
}
```

9.3.3.3 HierarchyGetProof()

This function finds the proof value associated with a hierarchy. It returns a pointer to the proof value.

```
TPM2B_AUTH *
HierarchyGetProof(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy constant
    )
{
    TPM2B_AUTH          *auth = NULL;

    switch(hierarchy)
    {
    case TPM_RH_PLATFORM:
        // phProof for TPM_RH_PLATFORM
        auth = &gp.phProof;
        break;
    case TPM_RH_ENDORSEMENT:
        // ehProof for TPM_RH_ENDORSEMENT
        auth = &gp.ehProof;
        break;
    case TPM_RH_OWNER:
        // shProof for TPM_RH_OWNER
        auth = &gp.shProof;
        break;
    case TPM_RH_NULL:
        // nullProof for TPM_RH_NULL
        auth = &gr.nullProof;
        break;
    default:
        pAssert(FALSE);
        break;
    }
    return auth;

}
```


9.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```
TPM2B_SEED *
HierarchyGetPrimarySeed(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy
    )
{
    TPM2B_SEED          *seed = NULL;
    switch(hierarchy)
    {
    case TPM_RH_PLATFORM:
```

```
            seed = &gp.PPSeed;
            break;
        case TPM_RH_OWNER:
            seed = &gp.SPSeed;
            break;
        case TPM_RH_ENDORSEMENT:
            seed = &gp.EPSeed;
            break;
        case TPM_RH_NULL:
            return &gr.nullSeed;
        default:
            pAssert(FALSE);
            break;
        }
        return seed;
    }
```

9.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE The TPM_RH_NULL hierarchy is always enabled.

Table 33

Return Value	Meaning
TRUE	hierarchy is enabled
FALSE	hierarchy is disabled

```
BOOL
HierarchyIsEnabled(
    TPMI_RH_HIERARCHY    hierarchy       // IN: hierarchy
    )
{
    BOOL            enabled = FALSE;

    switch(hierarchy)
    {
    case TPM_RH_PLATFORM:
        enabled = g_phEnable;
        break;
    case TPM_RH_OWNER:
        enabled = gc.shEnable;
        break;
    case TPM_RH_ENDORSEMENT:
        enabled = gc.ehEnable;
        break;
    case TPM_RH_NULL:
        enabled = TRUE;
        break;
    default:
        pAssert(FALSE);
        break;
    }
    return enabled;
}
```

9.4 NV.c

9.4.1 Introduction

The NV memory is divided into two area: dynamic space for user defined NV Indices and evict objects, and reserved space for TPM persistent and state save data.

9.4.2 Includes, Defines and Data Definitions

```
#define NV_C
#include "InternalRoutines.h"
#include <Platform.h>
```

NV Index/evict object iterator value

```
typedef     UINT32          NV_ITER;        // type of a NV iterator
#define     NV_ITER_INIT    0xFFFFFFFF      // initial value to start an
                                            // iterator
```

9.4.3 NV Utility Functions

9.4.3.1 NvCheckState()

Function to check the NV state by accessing the platform-specific function to get the NV state. The result state is registered in *s_NvIsAvailable* that will be reported by NvIsAvailable().

This function is called at the beginning of ExecuteCommand() before any potential call to NvIsAvailable().

```
void
NvCheckState(void)
{
    int     func_return;

    func_return = _plat__IsNvAvailable();
    if(func_return == 0)
    {
        s_NvStatus = TPM_RC_SUCCESS;
    }
    else if(func_return == 1)
    {
        s_NvStatus = TPM_RC_NV_UNAVAILABLE;
    }
    else
    {
        s_NvStatus = TPM_RC_NV_RATE;
    }

    return;
}
```

9.4.3.2 NvIsAvailable()

This function returns the NV availability parameter.

Table 34

Error Returns	Meaning
TPM_RC_SUCCESS	NV is available
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```
TPM_RC
NvIsAvailable(
    void
    )
{
    return s_NvStatus;
}
```

9.4.3.3 NvCommit

This is a wrapper for the platform function to commit pending NV writes.

```
BOOL
NvCommit(
    void
    )
{
    BOOL    success = (_plat__NvCommit() == 0);
    return success;
}
```

9.4.3.4 NvReadMaxCount()

This function returns the max NV counter value.

```
static UINT64
NvReadMaxCount(
    void
    )
{
    UINT64      countValue;
    _plat__NvMemoryRead(s_maxCountAddr, sizeof(UINT64), &countValue);
    return countValue;
}
```

9.4.3.5 NvWriteMaxCount()

This function updates the max counter value to NV memory.

```
static void
NvWriteMaxCount(
    UINT64           maxCount
    )
{
    _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &maxCount);
    return;
}
```

9.4.4 NV Index and Persistent Object Access Functions

9.4.4.1 Introduction

These functions are used to access an NV Index and persistent object memory. In this implementation, the memory is simulated with RAM. The data in dynamic area is organized as a linked list, starting from address *s_evictNvStart*. The first 4 bytes of a node in this link list is the offset of next node, followed by the data entry. A 0-valued offset value indicates the end of the list. If the data entry area of the last node happens to reach the end of the dynamic area without space left for an additional 4 byte end marker, the end address, *s_evictNvEnd*, should serve as the mark of list end.

9.4.4.2 NvNext()

This function provides a method to traverse every data entry in NV dynamic area.

To begin with, parameter *iter* should be initialized to NV_ITER_INIT indicating the first element. Every time this function is called, the value in *iter* would be adjusted pointing to the next element in traversal. If there is no next element, *iter* value would be 0. This function returns the address of the 'data entry' pointed by the *iter*. If there is no more element in the set, a 0 value is returned indicating the end of traversal.

```
static UINT32
NvNext(
    NV_ITER         *iter
    )
{
    NV_ITER     currentIter;

    // If iterator is at the beginning of list
    if(*iter == NV_ITER_INIT)
    {
        // Initialize iterator
        *iter = s_evictNvStart;
    }

    // If iterator reaches the end of NV space, or iterator indicates list end
    if(*iter + sizeof(UINT32) > s_evictNvEnd || *iter == 0)
        return 0;

    // Save the current iter offset
    currentIter = *iter;

    // Adjust iter pointer pointing to next entity
    // Read pointer value
    _plat__NvMemoryRead(*iter, sizeof(UINT32), iter);

    if(*iter == 0) return 0;

    return currentIter + sizeof(UINT32);      // entity stores after the pointer
}
```

9.4.4.3 NvGetEnd()

Function to find the end of the NV dynamic data list.

```
static UINT32
NvGetEnd(
    void
    )
{
```

```
    NV_ITER         iter = NV_ITER_INIT;
    UINT32          endAddr = s_evictNvStart;
    UINT32          currentAddr;

    while((currentAddr = NvNext(&iter)) != 0)
        endAddr = currentAddr;

    if(endAddr != s_evictNvStart)
    {
        // Read offset
        endAddr -= sizeof(UINT32);
        _plat__NvMemoryRead(endAddr, sizeof(UINT32), &endAddr);
    }

    return endAddr;
}
```

9.4.4.4 NvGetFreeByte

This function returns the number of free octets in NV space.

```
static UINT32
NvGetFreeByte(
    void
    )
{
    return s_evictNvEnd - NvGetEnd();
}
```

9.4.4.5 NvGetEvictObjectSize

This function returns the size of an evict object in NV space.

```
static UINT32
NvGetEvictObjectSize(
    void
    )
{
    return sizeof(TPM_HANDLE) + sizeof(OBJECT) + sizeof(UINT32);
}
```

9.4.4.6 NvGetCounterSize

This function returns the size of a counter index in NV space.

```
static UINT32
NvGetCounterSize(
    void
    )
{
    // It takes an offset field, a handle and the sizeof(NV_INDEX) and
    // sizeof(UINT64) for counter data
    return sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + sizeof(UINT64) + sizeof(UINT32);
}
```

9.4.4.7 NvTestSpace()

This function will test if there is enough space to add a new entity.

Table 35

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```
static BOOL
NvTestSpace(
    UINT32              size,           // IN: size of the entity to be added
    BOOL                isIndex         // IN: TRUE if the entity is an index
    )
{
    UINT32      remainByte = NvGetFreeByte();

    // For NV Index, need to make sure that we do not allocate and Index if this
    // would mean that the TPM cannot allocate the minimum number of evict
    // objects.
    if(isIndex)
    {
        // Get the number of persistent objects allocated
        UINT32      persistentNum = NvCapGetPersistentNumber();

        // If we have not allocated the requisite number of evict objects, then we
        // need to reserve space for them.
        // NOTE: some of this is not written as simply as it might seem because
        // the values are all unsigned and subtracting needs to be done carefully
        // so that an underflow doesn't cause problems.
        if(persistentNum < MIN_EVICT_OBJECTS)
        {
            UINT32      needed = (MIN_EVICT_OBJECTS - persistentNum)
                                * NvGetEvictObjectSize();
            if(needed > remainByte)
                remainByte = 0;
            else
                remainByte -= needed;
        }
        // if the requisite number of evict objects have been allocated then
        // no need to reserve additional space
    }
    // This checks for the size of the value being added plus the index value.
    // NOTE: This does not check to see if the end marker can be placed in
    // memory because the end marker will not be written if it will not fit.
    return (size + sizeof(UINT32) <= remainByte);
}
```

9.4.4.8 NvAdd()

This function adds a new entity to NV.

This function requires that there is enough space to add a new entity (i. e. , that NvTestSpace() has been called and the available space is at least as large as the required space).

```
static void
NvAdd(
    UINT32              totalSize,      // IN: total size needed for this entity For
                                        //     evict object, totalSize is the same as
                                        //     bufferSize.  For NV Index, totalSize is
                                        //     bufferSize plus index data size
    UINT32              bufferSize,     // IN: size of initial buffer
    BYTE               *entity          // IN: initial buffer
    )
{
```

```
    UINT32          endAddr;
    UINT32          nextAddr;
    UINT32          listEnd = 0;

    // Get the end of data list
    endAddr = NvGetEnd();

    // Calculate the value of next pointer, which is the size of a pointer +
    // the entity data size
    nextAddr = endAddr + sizeof(UINT32) + totalSize;

    // Write next pointer
    _plat__NvMemoryWrite(endAddr, sizeof(UINT32), &nextAddr);

    // Write entity data
    _plat__NvMemoryWrite(endAddr + sizeof(UINT32), bufferSize, entity);

    // Write the end of list if it is not going to exceed the NV space
    if(nextAddr + sizeof(UINT32) <= s_evictNvEnd)
        _plat__NvMemoryWrite(nextAddr, sizeof(UINT32), &listEnd);

    // Set the flag so that NV changes are committed before the command completes.
    g_updateNV = TRUE;
}
```

9.4.4.9 NvDelete()

This function is used to delete an NV Index or persistent object from NV memory.

```
static void
NvDelete(
    UINT32           entityAddr     // IN: address of entity to be deleted
    )
{
    UINT32          next;
    UINT32          entrySize;
    UINT32          entryAddr = entityAddr - sizeof(UINT32);
    UINT32          listEnd = 0;

    // Get the offset of the next entry.
    _plat__NvMemoryRead(entryAddr, sizeof(UINT32), &next);

    // The size of this entry is the difference between the current entry and the
    // next entry.
    entrySize = next - entryAddr;

    // Move each entry after the current one to fill the freed space.
    // Stop when we have reached the end of all the indexes. There are two
    // ways to detect the end of the list. The first is to notice that there
    // is no room for anything else because we are at the end of NV. The other
    // indication is that we find an end marker.

    // The loop condition checks for the end of NV.
    while(next + sizeof(UINT32) <= s_evictNvEnd)
    {
        UINT32      size, oldAddr, newAddr;

        // Now check for the end marker
        _plat__NvMemoryRead(next, sizeof(UINT32), &oldAddr);
        if(oldAddr == 0)
            break;

        size = oldAddr - next;

```

```
        // Move entry
        _plat__NvMemoryMove(next, next - entrySize, size);

        // Update forward link
        newAddr = oldAddr - entrySize;
        _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &newAddr);
        next = oldAddr;
    }
    // Mark the end of list
    _plat__NvMemoryWrite(next - entrySize, sizeof(UINT32), &listEnd);

    // Set the flag so that NV changes are committed before the command completes.
    g_updateNV = TRUE;
}
```

9.4.5 RAM-based NV Index Data Access Functions

9.4.5.1 Introduction

The data layout in ram buffer is {size of(*NV_handle*() + data), NV_handle(), data} for each NV Index data stored in RAM.

NV storage is updated when a NV Index is added or deleted. We do NOT updated NV storage when the data is updated.

9.4.5.2 NvTestRAMSpace()

This function indicates if there is enough RAM space to add a data for a new NV Index.

Table 36

Return Value	Meaning
TRUE	space available
FALSE	no enough space

```
static BOOL
NvTestRAMSpace(
    UINT32           size           // IN: size of the data to be added to RAM
    )
{
    BOOL         success = (   s_ramIndexSize
                             + size
                             + sizeof(TPM_HANDLE) + sizeof(UINT32)
                             <= RAM_INDEX_SPACE);
    return success;
}
```

9.4.5.3 NvGetRamIndexOffset

This function returns the offset of NV data in the RAM buffer.

This function requires that NV Index is in RAM. That is, the index must be known to exist.

```
static UINT32
NvGetRAMIndexOffset(
    TPMI_RH_NV_INDEX     handle         // IN: NV handle
    )
{
```



```
    UINT32      currAddr = 0;

    while(currAddr < s_ramIndexSize)
    {
        TPMI_RH_NV_INDEX    currHandle;
        UINT32              currSize;
        currHandle = * (TPM_HANDLE *) &s_ramIndex[currAddr + sizeof(UINT32)];

        // Found a match
        if(currHandle == handle)

            // data buffer follows the handle and size field
            break;

        currSize = * (UINT32 *) &s_ramIndex[currAddr];
        currAddr += sizeof(UINT32) + currSize;
    }

    // We assume the index data is existing in RAM space
    pAssert(currAddr < s_ramIndexSize);
    return currAddr + sizeof(TPMI_RH_NV_INDEX) + sizeof(UINT32);
}
```

9.4.5.4 NvAddRAM()

This function adds a new data area to RAM.

This function requires that enough free RAM space is available to add the new data.

```
static void
NvAddRAM(
    TPMI_RH_NV_INDEX    handle,         // IN: NV handle
    UINT32              size            // IN: size of data
    )
{
    // Add data space at the end of reserved RAM buffer
    * (UINT32 *) &s_ramIndex[s_ramIndexSize] = size + sizeof(TPMI_RH_NV_INDEX);
    * (TPMI_RH_NV_INDEX *) &s_ramIndex[s_ramIndexSize + sizeof(UINT32)] = handle;
    s_ramIndexSize += sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX) + size;

    pAssert(s_ramIndexSize <= RAM_INDEX_SPACE);

    // Update NV version of s_ramIndexSize
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);

    // Write reserved RAM space to NV to reflect the newly added NV Index
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    return;
}
```

9.4.5.5 NvDeleteRAM()

This function is used to delete a RAM-backed NV Index data area.

This function assumes the data of NV Index exists in RAM.

```
static void
NvDeleteRAM(
    TPMI_RH_NV_INDEX    handle          // IN: NV handle
    )
{
    UINT32          nodeOffset;
```

```
    UINT32          nextNode;
    UINT32          size;

    nodeOffset = NvGetRAMIndexOffset(handle);

    // Move the pointer back to get the size field of this node
    nodeOffset -= sizeof(UINT32) + sizeof(TPMI_RH_NV_INDEX);

    // Get node size
    size = * (UINT32 *) &s_ramIndex[nodeOffset];

    // Get the offset of next node
    nextNode = nodeOffset + sizeof(UINT32) + size;

    // Move data
    MemoryMove(s_ramIndex + nodeOffset, s_ramIndex + nextNode,
               s_ramIndexSize - nextNode, s_ramIndexSize - nextNode);

    // Update RAM size
    s_ramIndexSize -= size + sizeof(UINT32);

    // Update NV version of s_ramIndexSize
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);

    // Write reserved RAM space to NV to reflect the newly delete NV Index
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    return;
}
```

9.4.6 Utility Functions

9.4.6.1 NvInitStatic()

This function initializes the static variables used in the NV subsystem.

```
static void
NvInitStatic(
    void
    )
{
    UINT16      i;
    UINT32      reservedAddr;

    s_reservedSize[NV_DISABLE_CLEAR] = sizeof(gp.disableClear);
    s_reservedSize[NV_OWNER_ALG] = sizeof(gp.ownerAlg);
    s_reservedSize[NV_ENDORSEMENT_ALG] = sizeof(gp.endorsementAlg);
    s_reservedSize[NV_LOCKOUT_ALG] = sizeof(gp.lockoutAlg);
    s_reservedSize[NV_OWNER_POLICY] = sizeof(gp.ownerPolicy);
    s_reservedSize[NV_ENDORSEMENT_POLICY] = sizeof(gp.endorsementPolicy);
    s_reservedSize[NV_LOCKOUT_POLICY] = sizeof(gp.lockoutPolicy);
    s_reservedSize[NV_OWNER_AUTH] = sizeof(gp.ownerAuth);
    s_reservedSize[NV_ENDORSEMENT_AUTH] = sizeof(gp.endorsementAuth);
    s_reservedSize[NV_LOCKOUT_AUTH] = sizeof(gp.lockoutAuth);
    s_reservedSize[NV_EP_SEED] = sizeof(gp.EPSeed);
    s_reservedSize[NV_SP_SEED] = sizeof(gp.SPSeed);
    s_reservedSize[NV_PP_SEED] = sizeof(gp.PPSeed);
    s_reservedSize[NV_PH_PROOF] = sizeof(gp.phProof);
    s_reservedSize[NV_SH_PROOF] = sizeof(gp.shProof);
    s_reservedSize[NV_EH_PROOF] = sizeof(gp.ehProof);
    s_reservedSize[NV_TOTAL_RESET_COUNT] = sizeof(gp.totalResetCount);
    s_reservedSize[NV_RESET_COUNT] = sizeof(gp.resetCount);
    s_reservedSize[NV_PCR_POLICIES] = sizeof(gp.pcrPolicies);
```



```
    s_reservedSize[NV_PCR_ALLOCATED] = sizeof(gp.pcrAllocated);
    s_reservedSize[NV_PP_LIST] = sizeof(gp.ppList);
    s_reservedSize[NV_FAILED_TRIES] = sizeof(gp.failedTries);
    s_reservedSize[NV_MAX_TRIES] = sizeof(gp.maxTries);
    s_reservedSize[NV_RECOVERY_TIME] = sizeof(gp.recoveryTime);
    s_reservedSize[NV_LOCKOUT_RECOVERY] = sizeof(gp.lockoutRecovery);
    s_reservedSize[NV_LOCKOUT_AUTH_ENABLED] = sizeof(gp.lockOutAuthEnabled);
    s_reservedSize[NV_ORDERLY] = sizeof(gp.orderlyState);
    s_reservedSize[NV_AUDIT_COMMANDS] = sizeof(gp.auditComands);
    s_reservedSize[NV_AUDIT_HASH_ALG] = sizeof(gp.auditHashAlg);
    s_reservedSize[NV_AUDIT_COUNTER] = sizeof(gp.auditCounter);
    s_reservedSize[NV_ALGORITHM_SET] = sizeof(gp.algorithmSet);
    s_reservedSize[NV_FIRMWARE_V1] = sizeof(gp.firmwareV1);
    s_reservedSize[NV_FIRMWARE_V2] = sizeof(gp.firmwareV2);
    s_reservedSize[NV_ORDERLY_DATA] = sizeof(go);
    s_reservedSize[NV_STATE_CLEAR] = sizeof(gc);
    s_reservedSize[NV_STATE_RESET] = sizeof(gr);

    // Initialize reserved data address.  In this implementation, reserved data
    // is stored at the start of NV memory
    reservedAddr = 0;
    for(i = 0; i < NV_RESERVE_LAST; i++)
    {
        s_reservedAddr[i] = reservedAddr;
        reservedAddr += s_reservedSize[i];
    }

    // Initialize auxiliary variable space for index/evict implementation.
    // Auxiliary variables are stored after reserved data area
    // RAM index copy starts at the beginning
    s_ramIndexSizeAddr = reservedAddr;
    s_ramIndexAddr = s_ramIndexSizeAddr + sizeof(UINT32);

    // Maximum counter value
    s_maxCountAddr = s_ramIndexAddr + RAM_INDEX_SPACE;

    // dynamic memory start
    s_evictNvStart = s_maxCountAddr + sizeof(UINT64);

    // dynamic memory ends at the end of NV memory
    s_evictNvEnd = NV_MEMORY_SIZE;

    return;
}
```

9.4.6.2 NvInit()

This function initializes the NV system at pre-install time.

This function should only be called in a manufacturing environment or in a simulation.

The layout of NV memory space is an implementation choice.

```
void
NvInit(
    void
    )
{
    UINT32      nullPointer = 0;
    UINT64      zeroCounter = 0;

    // Initialize static variables
    NvInitStatic();

    // Initialize RAM index space as unused
```

```
    _plat__NvMemoryWrite(s_ramIndexSizeAddr, sizeof(UINT32), &nullPointer);

    // Initialize max counter value to 0
    _plat__NvMemoryWrite(s_maxCountAddr, sizeof(UINT64), &zeroCounter);

    // Initialize the next offset of the first entry in evict/index list to 0
    _plat__NvMemoryWrite(s_evictNvStart, sizeof(TPM_HANDLE), &nullPointer);

    return;

}
```

9.4.6.3 NvReadReserved()

This function is used to move reserved data from NV memory to RAM.

```
void
NvReadReserved(
    NV_RESERVE       type,          // IN: type of reserved data
    void            *buffer         // OUT: buffer receives the data.
    )
{
    // Input type should be valid
    pAssert(type >= 0 && type < NV_RESERVE_LAST);

    _plat__NvMemoryRead(s_reservedAddr[type], s_reservedSize[type], buffer);
    return;
}
```

9.4.6.4 NvWriteReserved()

This function is used to post a reserved data for writing to NV memory. Before the TPM completes the operation, the value will be written.

```
void
NvWriteReserved(
    NV_RESERVE       type,          // IN: type of reserved data
    void            *buffer         // IN: data buffer
    )
{
    // Input type should be valid
    pAssert(type >= 0 && type < NV_RESERVE_LAST);

    _plat__NvMemoryWrite(s_reservedAddr[type], s_reservedSize[type], buffer);

    // Set the flag that a NV write happens
    g_updateNV = TRUE;
    return;
}
```

9.4.6.5 NvReadPersistent()

This function reads persistent data to the RAM copy of the *gp* structure.

```
void
NvReadPersistent(
    void
    )
{
    // Hierarchy persistent data
    NvReadReserved(NV_DISABLE_CLEAR, &gp.disableClear);
```



```
    NvReadReserved(NV_OWNER_ALG, &gp.ownerAlg);
    NvReadReserved(NV_ENDORSEMENT_ALG, &gp.endorsementAlg);
    NvReadReserved(NV_LOCKOUT_ALG, &gp.lockoutAlg);
    NvReadReserved(NV_OWNER_POLICY, &gp.ownerPolicy);
    NvReadReserved(NV_ENDORSEMENT_POLICY, &gp.endorsementPolicy);
    NvReadReserved(NV_LOCKOUT_POLICY, &gp.lockoutPolicy);
    NvReadReserved(NV_OWNER_AUTH, &gp.ownerAuth);
    NvReadReserved(NV_ENDORSEMENT_AUTH, &gp.endorsementAuth);
    NvReadReserved(NV_LOCKOUT_AUTH, &gp.lockoutAuth);
    NvReadReserved(NV_EP_SEED, &gp.EPSeed);
    NvReadReserved(NV_SP_SEED, &gp.SPSeed);
    NvReadReserved(NV_PP_SEED, &gp.PPSeed);
    NvReadReserved(NV_PH_PROOF, &gp.phProof);
    NvReadReserved(NV_SH_PROOF, &gp.shProof);
    NvReadReserved(NV_EH_PROOF, &gp.ehProof);

    // Time persistent data
    NvReadReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);
    NvReadReserved(NV_RESET_COUNT, &gp.resetCount);

    // PCR persistent data
    NvReadReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
    NvReadReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);

    // Physical Presence persistent data
    NvReadReserved(NV_PP_LIST, &gp.ppList);

    // Dictionary attack values persistent data
    NvReadReserved(NV_FAILED_TRIES, &gp.failedTries);
    NvReadReserved(NV_MAX_TRIES, &gp.maxTries);
    NvReadReserved(NV_RECOVERY_TIME, &gp.recoveryTime);
    NvReadReserved(NV_LOCKOUT_RECOVERY, &gp.lockoutRecovery);
    NvReadReserved(NV_LOCKOUT_AUTH_ENABLED, &gp.lockOutAuthEnabled);

    // Orderly State persistent data
    NvReadReserved(NV_ORDERLY, &gp.orderlyState);

    // Command audit values persistent data
    NvReadReserved(NV_AUDIT_COMMANDS, &gp.auditComands);
    NvReadReserved(NV_AUDIT_HASH_ALG, &gp.auditHashAlg);
    NvReadReserved(NV_AUDIT_COUNTER, &gp.auditCounter);

    // Algorithm selection persistent data
    NvReadReserved(NV_ALGORITHM_SET, &gp.algorithmSet);

    // Firmware version persistent data
    NvReadReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
    NvReadReserved(NV_FIRMWARE_V2, &gp.firmwareV2);

    return;
}
```


9.4.6.6 NvIsPlatformPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the platform.

Table 37

Return Value	Meaning
TRUE	handle references a platform persistent object
FALSE	handle does not reference platform persistent object and may reference an owner persistent object either

```
BOOL
NvIsPlatformPersistentHandle(
    TPM_HANDLE       handle         // IN: handle
    )
{
    return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
}
```

9.4.6.7 NvIsOwnerPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the owner.

Table 38

Return Value	Meaning
TRUE	handle is owner persistent handle
FALSE	handle is not owner persistent handle and may not be a persistent handle at all

```
BOOL
NvIsOwnerPersistentHandle(
    TPM_HANDLE       handle         // IN: handle
    )
{
    return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
}
```

9.4.6.8 NvNextIndex()

This function returns the offset in NV of the next NV Index entry. A value of 0 indicates the end of the list.

```
static UINT32
NvNextIndex(
    NV_ITER          *iter
    )
{
    UINT32      addr;
    TPM_HANDLE  handle;

    while((addr = NvNext(iter)) != 0)
    {
        // Read handle
        _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
        if(HandleGetType(handle) == TPM_HT_NV_INDEX)
            return addr;
    }

    pAssert(addr == 0);
    return addr;
}
```

9.4.6.9 NvNextEvict()

This function returns the offset in NV of the next evict object entry. A value of 0 indicates the end of the list.

```
static UINT32
NvNextEvict(
    NV_ITER         *iter
    )
{
    UINT32      addr;
    TPM_HANDLE  handle;

    while((addr = NvNext(iter)) != 0)
    {
        // Read handle
        _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &handle);
        if(HandleGetType(handle) == TPM_HT_PERSISTENT)
            return addr;
    }

    pAssert(addr == 0);
    return addr;
}
```

9.4.6.10 NvFindHandle()

this function returns the offset in NV memory of the entity associated with the input handle. A value of zero indicates that handle does not exist reference an existing persistent object or defined NV Index.

```
static UINT32
NvFindHandle(
    TPM_HANDLE       handle
    )
{
    UINT32          addr;
    NV_ITER         iter = NV_ITER_INIT;

    while((addr = NvNext(&iter)) != 0)
    {
        TPM_HANDLE          entityHandle;
        // Read handle
        _plat__NvMemoryRead(addr, sizeof(TPM_HANDLE), &entityHandle);
        if(entityHandle == handle)
            return addr;
    }

    pAssert(addr == 0);
    return addr;
}
```

9.4.6.11 NvPowerOn()

This function is called at _TPM_Init() to initialize the NV environment.

Table 39

Return Value	Meaning
TRUE	all NV was initialized
FALSE	the NV containing saved state had an error and TPM2_Startup(CLEAR) is required

```
BOOL
NvPowerOn(
    void
    )
{
    int         nvError = 0;
    // If power was lost, need to re-establish the RAM data that is loaded from
    // NV and initialize the static variables
    if(_plat__WasPowerLost(TRUE))
    {
        if((nvError = _plat__NVEnable(0)) < 0)
            FAIL(FATAL_ERROR_NV_UNRECOVERABLE);

        NvInitStatic();
    }

    return nvError == 0;
}
```

9.4.6.12 NvStateSave()

This function is used to cause the memory containing the RAM backed NV Indices to be written to NV.

```
void
NvStateSave(
    void
    )
{
    // Write RAM backed NV Index info to NV
    // No need to save s_ramIndexSize because we save it to NV whenever it is
    // updated.
    _plat__NvMemoryWrite(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    // Set the flag so that an NV write happens before the command completes.
    g_updateNV = TRUE;

    return;
}
```

9.4.6.13 NvEntityStartup()

This function is called at TPM_Startup(). If the startup completes a TPM Resume cycle, no action is taken. If the startup is a TPM Reset or a TPM Restart, then this function will:

a) clear read/write lock;

b) reset NV Index data that has TPMA_NV_CLEAR_STCLEAR SET; and

c) set the lower bits in orderly counters to 1 for a non-orderly startup

It is a prerequisite that NV be available for writing before this function is called.

```
void
NvEntityStartup(
```



```
    STARTUP_TYPE     type           // IN: start up type
    )
{
    NV_ITER             iter = NV_ITER_INIT;
    UINT32              currentAddr;        // offset points to the current entity

    // Restore RAM index data
    _plat__NvMemoryRead(s_ramIndexSizeAddr, sizeof(UINT32), &s_ramIndexSize);
    _plat__NvMemoryRead(s_ramIndexAddr, RAM_INDEX_SPACE, s_ramIndex);

    // If recovering from state save, do nothing
    if(type == SU_RESUME)
        return;

    // Iterate all the NV Index to clear the locks
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        NV_INDEX    nvIndex;
        UINT32      indexAddr;              // NV address points to index info
        TPMA_NV     attributes;

        indexAddr = currentAddr + sizeof(TPM_HANDLE);

        // Read NV Index info structure
        _plat__NvMemoryRead(indexAddr, sizeof(NV_INDEX), &nvIndex);
        attributes = nvIndex.publicArea.attributes;

        // Clear read/write lock
        if(attributes.TPMA_NV_READLOCKED == SET)
            attributes.TPMA_NV_READLOCKED = CLEAR;

        if(     attributes.TPMA_NV_WRITELOCKED == SET
            &&  (   attributes.TPMA_NV_WRITTEN == CLEAR
                ||  attributes.TPMA_NV_WRITEDEFINE == CLEAR
                )
           )
            attributes.TPMA_NV_WRITELOCKED = CLEAR;

        // Reset NV data for TPMA_NV_CLEAR_STCLEAR
        if(attributes.TPMA_NV_CLEAR_STCLEAR == SET)
        {
            attributes.TPMA_NV_WRITTEN = CLEAR;
            attributes.TPMA_NV_WRITELOCKED = CLEAR;
        }

        // Reset NV data for orderly values that are not counters
        // NOTE: The function has already exited on a TPM Resume, so the only
        // things being processed are TPM Restart and TPM Reset
        if(     type == SU_RESET
            &&  attributes.TPMA_NV_ORDERLY == SET
            &&  attributes.TPMA_NV_COUNTER == CLEAR
           )
            attributes.TPMA_NV_WRITTEN = CLEAR;

        // Write NV Index info back if it has changed
        if(*((UINT32 *)&attributes) != *((UINT32 *)&nvIndex.publicArea.attributes))
        {
            nvIndex.publicArea.attributes = attributes;
            _plat__NvMemoryWrite(indexAddr, sizeof(NV_INDEX), &nvIndex);

            // Set the flag that a NV write happens
            g_updateNV = TRUE;
        }
        // Set the lower bits in an orderly counter to 1 for a non-orderly startup
        if(    g_prevOrderlyState == SHUTDOWN_NONE
           && attributes.TPMA_NV_WRITTEN == SET)
```

```
        {
            if(   attributes.TPMA_NV_ORDERLY == SET
               && attributes.TPMA_NV_COUNTER == SET)
            {
                TPMI_RH_NV_INDEX    nvHandle;
                UINT64              counter;

                // Read NV handle
                _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);

                // Read the counter value saved to NV upon the last roll over.
                // Do not use RAM backed storage for this once.
                nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = CLEAR;
                NvGetIntIndexData(nvHandle, &nvIndex, &counter);
                nvIndex.publicArea.attributes.TPMA_NV_ORDERLY = SET;

                // Set the lower bits of counter to 1's
                counter |= MAX_ORDERLY_COUNT;

                // Write back to RAM
                NvWriteIndexData(nvHandle, &nvIndex, 0, sizeof(counter), &counter);

                // No write to NV because an orderly shutdown will update the
                // counters.

            }
        }
    }

    return;

}
```

9.4.7 NV Access Functions

9.4.7.1 Introduction

This set of functions provide accessing NV Index and persistent objects based using a handle for reference to the entity.

9.4.7.2 NvIsUndefinedIndex()

This function is used to verify that an NV Index is not defined. This is only used by TPM2_NV_DefineSpace().

Table 40

Return Value	Meaning
TRUE	the handle points to an existing NV Index
FALSE	the handle points to a non-existent Index

```
BOOL
NvIsUndefinedIndex(
    TPMI_RH_NV_INDEX     handle          // IN: handle
    )
{
    UINT32          entityAddr;          // offset points to the entity

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
```

```

    // Find the address of index
    entityAddr = NvFindHandle(handle);

    // If handle is not found, return TPM_RC_SUCCESS
    if(entityAddr == 0)
        return TPM_RC_SUCCESS;

    // NV Index is defined
    return TPM_RC_NV_DEFINED;
}
```

9.4.7.3 NvIndexIsAccessible()

This function validates that a handle references a defined NV Index and that the Index is currently accessible.

Table 41

Error Returns	Meaning
TPM_RC_HANDLE	the handle points to an undefined NV Index If *shEnable* is CLEAR, this would include an index created using *ownerAuth*. If *phEnableNV* is CLEAR, this would include and index created using platform auth
TPM_RC_NV_READLOCKED	Index is present but locked for reading and command does not write to the index
TPM_RC_NV_WRITELOCKED	Index is present but locked for writing and command writes to the index

```
TPM_RC
NvIndexIsAccessible(
    TPMI_RH_NV_INDEX     handle,        // IN: handle
    TPM_CC               commandCode    // IN: the command
    )
{
    UINT32              entityAddr;     // offset points to the entity
    NV_INDEX            nvIndex;        //

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

    // Find the address of index
    entityAddr = NvFindHandle(handle);

    // If handle is not found, return TPM_RC_HANDLE
    if(entityAddr == 0)
        return TPM_RC_HANDLE;

    // Read NV Index info structure
    _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
                        &nvIndex);

    if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
    {
        // if shEnable is CLEAR, an ownerCreate NV Index should not be
        // indicated as present
        if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
        {
            if(gc.shEnable == FALSE)
                return TPM_RC_HANDLE;
        }
        // if phEnableNV is CLEAR, a platform created Index should not
        // be visible
```

```
        else if(gc.phEnableNV == FALSE)
            return TPM_RC_HANDLE;
    }

    // If the Index is write locked and this is an NV Write operation...
    if(     nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED
        &&  IsWriteOperation(commandCode))
    {
        // then return a locked indication unless the command is TPM2_NV_WriteLock
        if(commandCode != TPM_CC_NV_WriteLock)
            return TPM_RC_NV_LOCKED;
        return TPM_RC_SUCCESS;
    }
    // If the Index is read locked and this is an NV Read operation...
    if(     nvIndex.publicArea.attributes.TPMA_NV_READLOCKED
        && IsReadOperation(commandCode))
    {
        // then return a locked indication unless the command is TPM2_NV_ReadLock
        if(commandCode != TPM_CC_NV_ReadLock)
            return TPM_RC_NV_LOCKED;
        return TPM_RC_SUCCESS;
    }

    // NV Index is accessible
    return TPM_RC_SUCCESS;
}
```

9.4.7.4 NvIsUndefinedEvictHandle()

This function indicates if a handle does not reference an existing persistent object. This function requires that the handle be in the proper range for persistent objects.

Table 42

Return Value	Meaning
TRUE	handle does not reference an existing persistent object
FALSE	handle does reference an existing persistent object

```
static BOOL
NvIsUndefinedEvictHandle(
    TPM_HANDLE          handle          // IN: handle
    )
{
    UINT32              entityAddr;     // offset points to the entity
    pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);

    // Find the address of evict object
    entityAddr = NvFindHandle(handle);

    // If handle is not found, return TRUE
    if(entityAddr == 0)
        return TRUE;
    else
        return FALSE;
}
```

9.4.7.5 NvGetEvictObject()

This function is used to dereference an evict object handle and get a pointer to the object.

Table 43

Error Returns	Meaning
TPM_RC_HANDLE	the handle does not point to an existing persistent object

```
TPM_RC
NvGetEvictObject(
    TPM_HANDLE       handle,        // IN: handle
    OBJECT          *object         // OUT: object data
    )
{
    UINT32          entityAddr;         // offset points to the entity
    TPM_RC          result = TPM_RC_SUCCESS;

    pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);

    // Find the address of evict object
    entityAddr = NvFindHandle(handle);

    // If handle is not found, return an error
    if(entityAddr == 0)
        result = TPM_RC_HANDLE;
    else
        // Read evict object
        _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE),
                            sizeof(OBJECT),
                            object);

    // whether there is an error or not, make sure that the evict
    // status of the object is set so that the slot will get freed on exit
    object->attributes.evict = SET;

    return result;
}
```

9.4.7.6 NvGetIndexInfo()

This function is used to retrieve the contents of an NV Index.

An implementation is allowed to save the NV Index in a vendor-defined format. If the format is different from the default used by the reference code, then this function would be changed to reformat the data into the default format.

A prerequisite to calling this function is that the handle must be known to reference a defined NV Index.

```
void
NvGetIndexInfo(
    TPMI_RH_NV_INDEX     handle,        // IN: handle
    NV_INDEX            *nvIndex        // OUT: NV index structure
    )
{
    UINT32               entityAddr;    // offset points to the entity

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

    // Find the address of NV index
    entityAddr = NvFindHandle(handle);
    pAssert(entityAddr != 0);

    // This implementation uses the default format so just
    // read the data in
    _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
                        nvIndex);
```

```
    return;
}
```

9.4.7.7 NvInitialCounter()

This function returns the value to be used when a counter index is initialized. It will scan the NV counters and find the highest value in any active counter. It will use that value as the starting point. If there are no active counters, it will use the value of the previous largest counter.

```
UINT64
NvInitialCounter(
    void
    )
{
    UINT64          maxCount;
    NV_ITER         iter = NV_ITER_INIT;
    UINT32          currentAddr;

    // Read the maxCount value
    maxCount = NvReadMaxCount();

    // Iterate all existing counters
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        TPMI_RH_NV_INDEX    nvHandle;
        NV_INDEX            nvIndex;

        // Read NV handle
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &nvHandle);

        // Get NV Index
        NvGetIndexInfo(nvHandle, &nvIndex);
        if(    nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
            && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
        {
            UINT64      countValue;
            // Read counter value
            NvGetIntIndexData(nvHandle, &nvIndex, &countValue);
            if(countValue > maxCount)
                maxCount = countValue;
        }
    }
    // Initialize the new counter value to be maxCount + 1
    // A counter is only initialized the first time it is written. The
    // way to write a counter is with TPM2_NV_INCREMENT(). Since the
    // "initial" value of a defined counter is the largest count value that
    // may have existed in this index previously, then the first use would
    // add one to that value.
    return maxCount;
}
```


9.4.7.8 NvGetIndexData()

This function is used to access the data in an NV Index. The data is returned as a byte sequence. Since counter values are kept in native format, they are converted to canonical form before being returned.

This function requires that the NV Index be defined, and that the required data is within the data range. It also requires that TPMA_NV_WRITTEN of the Index is SET.

```
void
NvGetIndexData(
```

```
    TPMI_RH_NV_INDEX     handle,        // IN: handle
    NV_INDEX            *nvIndex,       // IN: RAM image of index header
    UINT32               offset,        // IN: offset of NV data
    UINT16               size,          // IN: size of NV data
    void                *data           // OUT: data buffer
    )
{

    pAssert(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET);

    if(   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
       || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET)
    {
        // Read bit or counter data in canonical form
        UINT64      dataInInt;
        NvGetIntIndexData(handle, nvIndex, &dataInInt);
        UINT64_TO_BYTE_ARRAY(dataInInt, (BYTE *)data);
    }
    else
    {
        if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
        {
            UINT32      ramAddr;

            // Get data from RAM buffer
            ramAddr = NvGetRAMIndexOffset(handle);
            MemoryCopy(data, s_ramIndex + ramAddr + offset, size, size);
        }
        else
        {
            UINT32      entityAddr;
            entityAddr = NvFindHandle(handle);
            // Get data from NV
            // Skip NV Index info, read data buffer
            entityAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;
            // Read the data
            _plat__NvMemoryRead(entityAddr, size, data);
        }
    }
    return;
}
```

9.4.7.9 NvGetIntIndexData()

Get data in integer format of a bit or counter NV Index.

This function requires that the NV Index is defined and that the NV Index previously has been written.

```
void
NvGetIntIndexData(
    TPMI_RH_NV_INDEX     handle,        // IN: handle
    NV_INDEX            *nvIndex,       // IN: RAM image of NV Index header
    UINT64              *data           // IN: UINT64 pointer for counter or bits
    )
{
    // Validate that index has been written and is the right type
    pAssert(   nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == SET
            && (   nvIndex->publicArea.attributes.TPMA_NV_BITS == SET
                || nvIndex->publicArea.attributes.TPMA_NV_COUNTER == SET
               )
           );

    // bit and counter value is store in native format for TPM CPU.  So we directly
    // copy the contents of NV to output data buffer
    if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
```

```
    {
        UINT32      ramAddr;

        // Get data from RAM buffer
        ramAddr = NvGetRAMIndexOffset(handle);
        MemoryCopy(data, s_ramIndex + ramAddr, sizeof(*data), sizeof(*data));
    }
    else
    {
        UINT32      entityAddr;
        entityAddr = NvFindHandle(handle);

        // Get data from NV
        // Skip NV Index info, read data buffer
        _plat__NvMemoryRead(
            entityAddr + sizeof(TPM_HANDLE) + sizeof(NV_INDEX),
            sizeof(UINT64), data);
    }

    return;
}
```

9.4.7.10 NvWriteIndexInfo()

This function is called to queue the write of NV Index data to persistent memory.

This function requires that NV Index is defined.

Table 44

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
TPM_RC
NvWriteIndexInfo(
    TPMI_RH_NV_INDEX     handle,          // IN: handle
    NV_INDEX            *nvIndex          // IN: NV Index info to be written
    )
{
    UINT32      entryAddr;
    TPM_RC      result;

    // Get the starting offset for the index in the RAM image of NV
    entryAddr = NvFindHandle(handle);
    pAssert(entryAddr != 0);

    // Step over the link value
    entryAddr = entryAddr + sizeof(TPM_HANDLE);

    // If the index data is actually changed, then a write to NV is required
    if(_plat__NvIsDifferent(entryAddr, sizeof(NV_INDEX),nvIndex))
    {
        // Make sure that NV is available
        result = NvIsAvailable();
        if(result != TPM_RC_SUCCESS)
            return result;
        _plat__NvMemoryWrite(entryAddr, sizeof(NV_INDEX), nvIndex);
        g_updateNV = TRUE;
    }
    return TPM_RC_SUCCESS;
}
```


9.4.7.11 NvWriteIndexData()

This function is used to write NV index data.

This function requires that the NV Index is defined, and the data is within the defined data range for the index.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```
TPM_RC
NvWriteIndexData(
    TPMI_RH_NV_INDEX     handle,        // IN: handle
    NV_INDEX            *nvIndex,       // IN: RAM copy of NV Index
    UINT32               offset,        // IN: offset of NV data
    UINT32               size,          // IN: size of NV data
    void                *data           // OUT: data buffer
    )
{
    TPM_RC               result;
    // Validate that write falls within range of the index
    pAssert(nvIndex->publicArea.dataSize >= offset + size);

    // Update TPMA_NV_WRITTEN bit if necessary
    if(nvIndex->publicArea.attributes.TPMA_NV_WRITTEN == CLEAR)
    {
        nvIndex->publicArea.attributes.TPMA_NV_WRITTEN = SET;
        result = NvWriteIndexInfo(handle, nvIndex);
        if(result != TPM_RC_SUCCESS)
            return result;
    }

    // Check to see if process for an orderly index is required.
    if(nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == SET)
    {
        UINT32      ramAddr;

        // Write data to RAM buffer
        ramAddr = NvGetRAMIndexOffset(handle);
        MemoryCopy(s_ramIndex + ramAddr + offset, data, size,
                   sizeof(s_ramIndex) - ramAddr - offset);

        // NV update does not happen for orderly index.  Have
        // to clear orderlyState to reflect that we have changed the
        // NV and an orderly shutdown is required. Only going to do this if we
        // are not processing a counter that has just rolled over
        if(g_updateNV == FALSE)
            g_clearOrderly = TRUE;
    }
    // Need to process this part if the Index isn't orderly or if it is
    // an orderly counter that just rolled over.
    if(g_updateNV || nvIndex->publicArea.attributes.TPMA_NV_ORDERLY == CLEAR)
    {
        // Processing for an index with TPMA_NV_ORDERLY CLEAR
        UINT32      entryAddr = NvFindHandle(handle);

        pAssert(entryAddr != 0);

        // Offset into the index to the first byte of the data to be written
        entryAddr += sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + offset;

        // If the data is actually changed, then a write to NV is required
```

```
        if(_plat__NvIsDifferent(entryAddr, size, data))
        {
            // Make sure that NV is available
            result = NvIsAvailable();
            if(result != TPM_RC_SUCCESS)
                return result;
            _plat__NvMemoryWrite(entryAddr, size, data);
            g_updateNV = TRUE;
        }
    }
    return TPM_RC_SUCCESS;
}
```

9.4.7.12 NvGetName()

This function is used to compute the Name of an NV Index.

The *name* buffer receives the bytes of the Name and the return value is the number of octets in the Name.

This function requires that the NV Index is defined.

```
UINT16
NvGetName(
    TPMI_RH_NV_INDEX     handle,         // IN: handle of the index
    NAME                *name            // OUT: name of the index
    )
{
    UINT16               dataSize, digestSize;
    NV_INDEX             nvIndex;
    BYTE                 marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
    BYTE                *buffer;
    HASH_STATE           hashState;

    // Get NV public info
    NvGetIndexInfo(handle, &nvIndex);

    // Marshal public area
    buffer = marshalBuffer;
    dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex.publicArea, &buffer, NULL);

    // hash public area
    digestSize = CryptStartHash(nvIndex.publicArea.nameAlg, &hashState);
    CryptUpdateDigest(&hashState, dataSize, marshalBuffer);

    // Complete digest leaving room for the nameAlg
    CryptCompleteHash(&hashState, digestSize, &((BYTE *)name)[2]);

    // Include the nameAlg
    UINT16_TO_BYTE_ARRAY(nvIndex.publicArea.nameAlg, (BYTE *)name);
    return digestSize + 2;
}
```

9.4.7.13 NvDefineIndex()

This function is used to assign NV memory to an NV Index.

Table 45

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```
TPM_RC
NvDefineIndex(
    TPMS_NV_PUBLIC   *publicArea,    // IN: A template for an area to create.
    TPM2B_AUTH       *authValue      // IN: The initial authorization value
    )
{
    // The buffer to be written to NV memory
    BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(NV_INDEX)];

    NV_INDEX        *nvIndex;           // a pointer to the NV_INDEX data in
                                        //   nvBuffer
    UINT16          entrySize;          // size of entry

    entrySize = sizeof(TPM_HANDLE) + sizeof(NV_INDEX) + publicArea->dataSize;

    // Check if we have enough space to create the NV Index
    // In this implementation, the only resource limitation is the available NV
    // space.  Other implementation may have other limitation on counter or on
    // NV slot
    if(!NvTestSpace(entrySize, TRUE)) return TPM_RC_NV_SPACE;

    // if the index to be defined is RAM backed, check RAM space availability
    // as well
    if(publicArea->attributes.TPMA_NV_ORDERLY == SET
            && !NvTestRAMSpace(publicArea->dataSize))
        return TPM_RC_NV_SPACE;


    // Copy input value to nvBuffer
        // Copy handle
    * (TPM_HANDLE *) nvBuffer = publicArea->nvIndex;

        // Copy NV_INDEX
    nvIndex = (NV_INDEX *) (nvBuffer + sizeof(TPM_HANDLE));
    nvIndex->publicArea = *publicArea;
    nvIndex->authValue = *authValue;

    // Add index to NV memory
    NvAdd(entrySize, sizeof(TPM_HANDLE) + sizeof(NV_INDEX), nvBuffer);

    // If the data of NV Index is RAM backed, add the data area in RAM as well
    if(publicArea->attributes.TPMA_NV_ORDERLY == SET)
        NvAddRAM(publicArea->nvIndex, publicArea->dataSize);

    return TPM_RC_SUCCESS;
}
```

9.4.7.14 NvAddEvictObject()

This function is used to assign NV memory to a persistent object.

Table 46

Error Returns	Meaning
TPM_RC_NV_HANDLE	the requested handle is already in use
TPM_RC_NV_SPACE	insufficient NV space

```
TPM_RC
NvAddEvictObject(
    TPMI_DH_OBJECT   evictHandle,    // IN: new evict handle
    OBJECT          *object          // IN: object to be added
    )
{
    // The buffer to be written to NV memory
    BYTE            nvBuffer[sizeof(TPM_HANDLE) + sizeof(OBJECT)];

    OBJECT          *nvObject;          // a pointer to the OBJECT data in
                                        // nvBuffer
    UINT16          entrySize;          // size of entry

    // evict handle type should match the object hierarchy
    pAssert(   (    NvIsPlatformPersistentHandle(evictHandle)
                && object->attributes.ppsHierarchy == SET)
            || (    NvIsOwnerPersistentHandle(evictHandle)
                && (    object->attributes.spsHierarchy == SET
                    || object->attributes.epsHierarchy == SET)));

    // An evict needs 4 bytes of handle + sizeof OBJECT
    entrySize = sizeof(TPM_HANDLE) + sizeof(OBJECT);

    // Check if we have enough space to add the evict object
    // An evict object needs 8 bytes in index table + sizeof OBJECT
    // In this implementation, the only resource limitation is the available NV
    // space.  Other implementation may have other limitation on evict object
    // handle space
    if(!NvTestSpace(entrySize, FALSE)) return TPM_RC_NV_SPACE;

    // Allocate a new evict handle
    if(!NvIsUndefinedEvictHandle(evictHandle))
        return TPM_RC_NV_DEFINED;

    // Copy evict object to nvBuffer
        // Copy handle
    * (TPM_HANDLE *) nvBuffer = evictHandle;

        // Copy OBJECT
    nvObject = (OBJECT *) (nvBuffer + sizeof(TPM_HANDLE));
    *nvObject = *object;

    // Set evict attribute and handle
    nvObject->attributes.evict = SET;
    nvObject->evictHandle = evictHandle;

    // Add evict to NV memory
    NvAdd(entrySize, entrySize, nvBuffer);

    return TPM_RC_SUCCESS;

}
```


9.4.7.15 NvDeleteEntity()

This function will delete a NV Index or an evict object.

This function requires that the index/evict object has been defined.

```
void
NvDeleteEntity(
    TPM_HANDLE       handle         // IN: handle of entity to be deleted
    )
{
    UINT32      entityAddr;     // pointer to entity

    entityAddr = NvFindHandle(handle);
    pAssert(entityAddr != 0);

    if(HandleGetType(handle) == TPM_HT_NV_INDEX)
    {
        NV_INDEX    nvIndex;

        // Read the NV Index info
        _plat__NvMemoryRead(entityAddr + sizeof(TPM_HANDLE), sizeof(NV_INDEX),
                            &nvIndex);

        // If the entity to be deleted is a counter with the maximum counter
        // value, record it in NV memory
        if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET
                && nvIndex.publicArea.attributes.TPMA_NV_WRITTEN == SET)
        {
            UINT64      countValue;
            UINT64      maxCount;
            NvGetIntIndexData(handle, &nvIndex, &countValue);
            maxCount = NvReadMaxCount();
            if(countValue > maxCount)
                NvWriteMaxCount(countValue);
        }
        // If the NV Index is RAM back, delete the RAM data as well
        if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
            NvDeleteRAM(handle);
    }
    NvDelete(entityAddr);

    return;

}
```

9.4.7.16 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated If the storage hierarchy is selected, the function will also delete any NV Index define using *ownerAuth*.

```
void
NvFlushHierarchy(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flushed.
    )
{
    NV_ITER          iter = NV_ITER_INIT;
    UINT32           currentAddr;

    while((currentAddr = NvNext(&iter)) != 0)
    {
        TPM_HANDLE      entityHandle;

        // Read handle information.
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);

        if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
        {
```

```
            // Handle NV Index
            NV_INDEX    nvIndex;

            // If flush endorsement or platform hierarchy, no NV Index would be
            // flushed
            if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
                continue;
            _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
                                sizeof(NV_INDEX), &nvIndex);

            // For storage hierarchy, flush OwnerCreated index
            if(  nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == CLEAR)
            {
                // Delete the NV Index
                NvDelete(currentAddr);

                // Re-iterate from beginning after a delete
                iter = NV_ITER_INIT;

                // If the NV Index is RAM back, delete the RAM data as well
                if(nvIndex.publicArea.attributes.TPMA_NV_ORDERLY == SET)
                    NvDeleteRAM(entityHandle);
            }
        }
        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
        {
            OBJECT          object;

            // Get evict object
            NvGetEvictObject(entityHandle, &object);

            // If the evict object belongs to the hierarchy to be flushed
            if(   (    hierarchy == TPM_RH_PLATFORM
                   &&  object.attributes.ppsHierarchy == SET)
               || (    hierarchy == TPM_RH_OWNER
                   &&  object.attributes.spsHierarchy == SET)
               || (    hierarchy == TPM_RH_ENDORSEMENT
                   &&  object.attributes.epsHierarchy == SET)
               )
            {
                // Delete the evict object
                NvDelete(currentAddr);

                // Re-iterate from beginning after a delete
                iter = NV_ITER_INIT;
            }
        }
        else
        {
            pAssert(FALSE);
        }
    }

    return;
}
```

9.4.7.17 NvSetGlobalLock()

This function is used to SET the TPMA_NV_WRITELOCKED attribute for all NV Indices that have TPMA_NV_GLOBALLOCK SET. This function is use by TPM2_NV_GlobalWriteLock().

```
void
NvSetGlobalLock(
    void
```



```
    )
{
    NV_ITER         iter = NV_ITER_INIT;
    UINT32          currentAddr;

    // Check all Indices
    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        NV_INDEX    nvIndex;

        // Read the index data
        _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
                            sizeof(NV_INDEX), &nvIndex);

        // See if it should be locked
        if(nvIndex.publicArea.attributes.TPMA_NV_GLOBALLOCK == SET)
        {

            // if so, lock it
            nvIndex.publicArea.attributes.TPMA_NV_WRITELOCKED = SET;

            _plat__NvMemoryWrite(currentAddr + sizeof(TPM_HANDLE),
                                 sizeof(NV_INDEX), &nvIndex);
            // Set the flag that a NV write happens
            g_updateNV = TRUE;
        }
    }

    return;

}
```

9.4.7.18 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES.

```
static void
InsertSort(
    TPML_HANDLE     *handleList,    // IN/OUT: sorted handle list
    UINT32           count,         // IN: maximum count in the handle list
    TPM_HANDLE       entityHandle   // IN: handle to be inserted
    )
{
    UINT32          i, j;
    UINT32          originalCount;

    // For a corner case that the maximum count is 0, do nothing
    if(count == 0) return;

    // For empty list, add the handle at the beginning and return
    if(handleList->count == 0)
    {
        handleList->handle[0] = entityHandle;
        handleList->count++;
        return;
    }

    // Check if the maximum of the list has been reached
    originalCount = handleList->count;
    if(originalCount < count)
        handleList->count++;

    // Insert the handle to the list
```

```
    for(i = 0; i < originalCount; i++)
    {
        if(handleList->handle[i] > entityHandle)
        {
            for(j = handleList->count - 1; j > i; j--)
            {
                handleList->handle[j] = handleList->handle[j-1];
            }
            break;
        }
    }

    // If a slot was found, insert the handle in this position
    if(i < originalCount || handleList->count > originalCount)
        handleList->handle[i] = entityHandle;

    return;
}
```

9.4.7.19 NvCapGetPersistent()

This function is used to get a list of handles of the persistent objects, starting at *handle*.

Handle must be in valid persistent object handle range, but does not have to reference an existing persistent object.

Table 47

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
NvCapGetPersistent(
    TPMI_DH_OBJECT    handle,         // IN: start handle
    UINT32            count,          // IN: maximum number of returned handles
    TPML_HANDLE      *handleList      // OUT: list of handle
    )
{
    TPMI_YES_NO          more = NO;
    NV_ITER              iter = NV_ITER_INIT;
    UINT32               currentAddr;

    pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    while((currentAddr = NvNextEvict(&iter)) != 0)
    {
        TPM_HANDLE      entityHandle;

        // Read handle information.
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);

        // Ignore persistent handles that have values less than the input handle
        if(entityHandle < handle)
            continue;

```



```
        // if the handles in the list have reached the requested count, and there
        // are still handles need to be inserted, indicate that there are more.
        if(handleList->count == count)
            more = YES;

        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list.  Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV Indices is small.  For an implementation that may allow
        // large number of NV Indices, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, entityHandle);

    }
    return more;
}
```

9.4.7.20 NvCapGetIndex()

This function returns a list of handles of NV Indices, starting from *handle*. *Handle* must be in the range of NV Indices, but does not have to reference an existing NV Index.

Table 48

Return Value	Meaning
YES	if there are more handles to report
NO	all the available handles has been reported

```
TPMI_YES_NO
NvCapGetIndex(
    TPMI_DH_OBJECT     handle,         // IN: start handle
    UINT32             count,          // IN: maximum number of returned handles
    TPML_HANDLE       *handleList      // OUT: list of handle
    )
{
    TPMI_YES_NO            more = NO;
    NV_ITER                iter = NV_ITER_INIT;
    UINT32                 currentAddr;

    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        TPM_HANDLE      entityHandle;

        // Read handle information.
        _plat__NvMemoryRead(currentAddr, sizeof(TPM_HANDLE), &entityHandle);

        // Ignore index handles that have values less than the 'handle'
        if(entityHandle < handle)
            continue;

        // if the count of handles in the list has reached the requested count,
        // and there are still handles to report, set more.
        if(handleList->count == count)
            more = YES;
```

```
        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list.  Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV Indices is small.  For an implementation that may allow
        // large number of NV Indices, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, entityHandle);
    }
    return more;
}
```

9.4.7.21 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```
UINT32
NvCapGetIndexNumber(
    void
    )
{
    UINT32          num = 0;
    NV_ITER         iter = NV_ITER_INIT;

    while(NvNextIndex(&iter) != 0) num++;

    return num;
}
```

9.4.7.22 NvCapGetPersistentNumber()

Function returns the count of persistent objects currently in NV memory.

```
UINT32
NvCapGetPersistentNumber(
    void
    )
{
    UINT32          num = 0;
    NV_ITER         iter = NV_ITER_INIT;

    while(NvNextEvict(&iter) != 0) num++;

    return num;
}
```

9.4.7.23 NvCapGetPersistentAvail()

This function returns an estimate of the number of additional persistent objects that could be loaded into NV memory.

```
UINT32
NvCapGetPersistentAvail(
    void
    )
{
    UINT32          availSpace;
    UINT32          objectSpace;

    // Compute the available space in NV storage
    availSpace = NvGetFreeByte();
```



```

    // Get the space needed to add a persistent object to NV storage
    objectSpace = NvGetEvictObjectSize();

    return availSpace / objectSpace;
}
```

9.4.7.24 NvCapGetCounterNumber()

Get the number of defined NV Indexes that have NV TPMA_NV_COUNTER attribute SET.

```
UINT32
NvCapGetCounterNumber(
    void
    )
{
    NV_ITER         iter = NV_ITER_INIT;
    UINT32          currentAddr;
    UINT32          num = 0;

    while((currentAddr = NvNextIndex(&iter)) != 0)
    {
        NV_INDEX    nvIndex;

        // Get NV Index info
        _plat__NvMemoryRead(currentAddr + sizeof(TPM_HANDLE),
                            sizeof(NV_INDEX), &nvIndex);
        if(nvIndex.publicArea.attributes.TPMA_NV_COUNTER == SET) num++;
    }

    return num;
}
```

9.4.7.25 NvCapGetCounterAvail()

This function returns an estimate of the number of additional counter type NV Indices that can be defined.

```
UINT32
NvCapGetCounterAvail(
    void
    )
{
    UINT32          availNVSpace;
    UINT32          availRAMSpace;
    UINT32          counterNVSpace;
    UINT32          counterRAMSpace;
    UINT32          persistentNum = NvCapGetPersistentNumber();

    // Get the available space in NV storage
    availNVSpace = NvGetFreeByte();

    if (persistentNum < MIN_EVICT_OBJECTS)
    {
        // Some space have to be reserved for evict object. Adjust availNVSpace.
        UINT32      reserved = (MIN_EVICT_OBJECTS - persistentNum)
                              * NvGetEvictObjectSize();
        if (reserved > availNVSpace)
            availNVSpace = 0;
        else
            availNVSpace -= reserved;
    }

    // Get the space needed to add a counter index to NV storage
```

```
    counterNVSpace = NvGetCounterSize();

    // Compute the available space in RAM
    availRAMSpace = RAM_INDEX_SPACE - s_ramIndexSize;

    // Compute the space needed to add a counter index to RAM storage
    // It takes an size field, a handle and sizeof(UINT64) for counter data
    counterRAMSpace = sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(UINT64);

    // Return the min of counter number in NV and in RAM
    if(availNVSpace / counterNVSpace > availRAMSpace / counterRAMSpace)
        return availRAMSpace / counterRAMSpace;
    else
        return availNVSpace / counterNVSpace;
}
```

9.5 Object.c

9.5.1 Introduction

This file contains the functions that manage the object store of the TPM.

9.5.2 Includes and Data Definitions

```
#define OBJECT_C
#include "InternalRoutines.h"
#include <Platform.h>
```

9.5.3 Functions

9.5.3.1 ObjectStartup()

This function is called at TPM2_Startup() to initialize the object subsystem.

```
void
ObjectStartup(
    void
    )
{
    UINT32      i;

    // object slots initialization
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        //Set the slot to not occupied
        s_objects[i].occupied = FALSE;
    }
    return;
}
```

9.5.3.2 ObjectCleanupEvict()

In this implementation, a persistent object is moved from NV into an object slot for processing. It is flushed after command execution. This function is called from ExecuteCommand().

```
void
ObjectCleanupEvict(
    void
    )
```



```
{
    UINT32      i;

    // This has to be iterated because a command may have two handles
    // and they may both be persistent.
    // This could be made to be more efficient so that a search is not needed.
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        // If an object is a temporary evict object, flush it from slot
        if(s_objects[i].object.entity.attributes.evict == SET)
            s_objects[i].occupied = FALSE;
    }

    return;
}
```

9.5.3.3 ObjectIsPresent()

This function checks to see if a transient handle references a loaded object. This routine should not be called if the handle is not a transient handle. The function validates that the handle is in the implementation-dependent allowed in range for loaded transient objects.

Table 49

Return Value	Meaning
TRUE	if the handle references a loaded object
FALSE	if the handle is not an object handle, or it does not reference to a loaded object

```
BOOL
ObjectIsPresent(
    TPMI_DH_OBJECT    handle          // IN: handle to be checked
    )
{
    UINT32          slotIndex;        // index of object slot

    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);

    // The index in the loaded object array is found by subtracting the first
    // object handle number from the input handle number. If the indicated
    // slot is occupied, then indicate that there is already is a loaded
    //  object associated with the handle.
    slotIndex = handle - TRANSIENT_FIRST;
    if(slotIndex >= MAX_LOADED_OBJECTS)
        return FALSE;

    return s_objects[slotIndex].occupied;
}
```

9.5.3.4 ObjectIsSequence()

This function is used to check if the object is a sequence object. This function should not be called if the handle does not reference a loaded object.

Table 50

Return Value	Meaning
TRUE	object is an HMAC, hash, or event sequence object
FALSE	object is not an HMAC, hash, or event sequence object

```
BOOL
ObjectIsSequence(
    OBJECT          *object         // IN: handle to be checked
    )
{
    pAssert (object != NULL);
    if(   object->attributes.hmacSeq == SET
       || object->attributes.hashSeq == SET
       || object->attributes.eventSeq == SET)
        return TRUE;
    else
        return FALSE;
}
```

9.5.3.5 ObjectGet()

This function is used to find the object structure associated with a handle.

This function requires that *handle* references a loaded object.

```
OBJECT*
ObjectGet(
    TPMI_DH_OBJECT   handle         // IN: handle of the object
    )
{
    pAssert(   handle >= TRANSIENT_FIRST
            && handle - TRANSIENT_FIRST < MAX_LOADED_OBJECTS);
    pAssert(s_objects[handle - TRANSIENT_FIRST].occupied == TRUE);

    // In this implementation, the handle is determined by the slot occupied by the
    // object.
    return &s_objects[handle - TRANSIENT_FIRST].object.entity;
}
```

9.5.3.6 ObjectGetName()

This function is used to access the Name of the object. In this implementation, the Name is computed when the object is loaded and is saved in the internal representation of the object. This function copies the Name data from the object into the buffer at *name* and returns the number of octets copied.

This function requires that *handle* references a loaded object.

```
UINT16
ObjectGetName(
    TPMI_DH_OBJECT   handle,        // IN: handle of the object
    NAME            *name           // OUT: name of the object
    )
{
    OBJECT      *object = ObjectGet(handle);
    if(object->publicArea.nameAlg == TPM_ALG_NULL)
        return 0;

    // Copy the Name data to the output
    MemoryCopy(name, object->name.t.name, object->name.t.size, sizeof(NAME));
    return  object->name.t.size;
```



```
}
```

9.5.3.7 ObjectGetNameAlg()

This function is used to get the Name algorithm of a object.

This function requires that *handle* references a loaded object.

```
TPMI_ALG_HASH
ObjectGetNameAlg(
    TPMI_DH_OBJECT    handle          // IN: handle of the object
    )
{
    OBJECT              *object = ObjectGet(handle);

    return object->publicArea.nameAlg;
}
```

9.5.3.8 ObjectGetQualifiedName()

This function returns the Qualified Name of the object. In this implementation, the Qualified Name is computed when the object is loaded and is saved in the internal representation of the object. The alternative would be to retain the Name of the parent and compute the QN when needed. This would take the same amount of space so it is not recommended that the alternate be used.

This function requires that *handle* references a loaded object.

```
void
ObjectGetQualifiedName(
    TPMI_DH_OBJECT    handle,         // IN: handle of the object
    TPM2B_NAME       *qualifiedName   // OUT: qualified name of the object
    )
{
    OBJECT      *object = ObjectGet(handle);
    if(object->publicArea.nameAlg == TPM_ALG_NULL)
        qualifiedName->t.size = 0;
    else
        // Copy the name
        *qualifiedName = object->qualifiedName;

    return;
}
```

9.5.3.9 ObjectDataGetHierarchy()

This function returns the handle for the hierarchy of an object.

```
TPMI_RH_HIERARCHY
ObjectDataGetHierarchy(
    OBJECT           *object          // IN :object
    )
{
    if(object->attributes.spsHierarchy)
    {
        return TPM_RH_OWNER;
    }
    else if(object->attributes.epsHierarchy)
    {
        return TPM_RH_ENDORSEMENT;
    }
    else if(object->attributes.ppsHierarchy)
```

```
        {
            return TPM_RH_PLATFORM;
        }
        else
        {
            return TPM_RH_NULL;
        }

}
```

9.5.3.10 ObjectGetHierarchy()

This function returns the handle of the hierarchy to which a handle belongs. This function is similar to ObjectDataGetHierarchy() but this routine takes a handle but ObjectDataGetHierarchy() takes an pointer to an object.

This function requires that *handle* references a loaded object.

```
TPMI_RH_HIERARCHY
ObjectGetHierarchy(
    TPMI_DH_OBJECT    handle         // IN :object handle
    )
{
    OBJECT           *object = ObjectGet(handle);

    return ObjectDataGetHierarchy(object);
}
```

9.5.3.11 ObjectAllocateSlot()

This function is used to allocate a slot in internal object array.

Table 51

Return Value	Meaning
TRUE	allocate success
FALSE	do not have free slot

```
static BOOL
ObjectAllocateSlot(
    TPMI_DH_OBJECT   *handle,        // OUT: handle of allocated object
    OBJECT           **object        // OUT: points to the allocated object
    )
{
    UINT32        i;

    // find an unoccupied handle slot
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(!s_objects[i].occupied)          // If found a free slot
        {
            // Mark the slot as occupied
            s_objects[i].occupied = TRUE;
            break;
        }
    }
    // If we reach the end of object slot without finding a free one, return
    // error.
    if(i == MAX_LOADED_OBJECTS) return FALSE;

```



```
        *handle = i + TRANSIENT_FIRST;
        *object = &s_objects[i].object.entity;

        // Initialize the object attributes
        MemorySet(&((*object)->attributes), 0, sizeof(OBJECT_ATTRIBUTES));

        return TRUE;
    }
```

9.5.3.12 ObjectLoad()

This function loads an object into an internal object structure. If an error is returned, the internal state is unchanged.

Table 52

Error Returns	Meaning
TPM_RC_BINDING	if the public and sensitive parts of the object are not matched
TPM_RC_KEY	if the parameters in the public area of the object are not consistent
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object
TPM_RC_TYPE	the public and private parts are not the same type

```
TPM_RC
ObjectLoad(
    TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy to which the object belongs
    TPMT_PUBLIC         *publicArea,     // IN: public area
    TPMT_SENSITIVE      *sensitive,      // IN: sensitive area (may be null)
    TPM2B_NAME          *name,           // IN: object's name (may be null)
    TPM_HANDLE           parentHandle,   // IN: handle of parent
    BOOL                 skipChecks,     // IN: flag to indicate if it is OK to skip
                                         //     consistency checks.
    TPMI_DH_OBJECT      *handle          // OUT: object handle
    )
{
    OBJECT               *object = NULL;
    OBJECT               *parent = NULL;
    TPM_RC                result = TPM_RC_SUCCESS;
    TPM2B_NAME            parentQN;         // Parent qualified name

    // Try to allocate a slot for new object
    if(!ObjectAllocateSlot(handle, &object))
        return TPM_RC_OBJECT_MEMORY;

    // Initialize public
    object->publicArea = *publicArea;
    if(sensitive != NULL)
        object->sensitive = *sensitive;

    // Are the consistency checks needed
    if(!skipChecks)
    {
        // Check if key size matches
        if(!CryptObjectIsPublicConsistent(&object->publicArea))
        {
            result = TPM_RC_KEY;
            goto ErrorExit;
        }
        if(sensitive != NULL)
        {
            // Check if public type matches sensitive type
```

```
            result = CryptObjectPublicPrivateMatch(object);
            if(result != TPM_RC_SUCCESS)
                goto ErrorExit;
        }
    }
    object->attributes.publicOnly = (sensitive == NULL);

    // If 'name' is NULL, then there is nothing left to do for this
    // object as it has no qualified name and it is not a member of any
    // hierarchy and it is temporary
    if(name == NULL || name->t.size == 0)
    {
        object->qualifiedName.t.size = 0;
        object->name.t.size = 0;
        object->attributes.temporary = SET;
        return TPM_RC_SUCCESS;
    }
    // If parent handle is a permanent handle, it is a primary or temporary
    // object
    if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
    {
        // initialize QN
        parentQN.t.size = 4;

        // for a primary key, parent qualified name is the handle of hierarchy
        UINT32_TO_BYTE_ARRAY(parentHandle, parentQN.t.name);
    }
    else
    {
        // Get hierarchy and qualified name of parent
        ObjectGetQualifiedName(parentHandle, &parentQN);

        // Check for stClear object
        parent = ObjectGet(parentHandle);
        if(   publicArea->objectAttributes.stClear == SET
           || parent->attributes.stClear == SET)
            object->attributes.stClear = SET;

    }
    object->name = *name;

    // Compute object qualified name
    ObjectComputeQualifiedName(&parentQN, publicArea->nameAlg,
                               name, &object->qualifiedName);

    // Any object in TPM_RH_NULL hierarchy is temporary
    if(hierarchy == TPM_RH_NULL)
    {
        object->attributes.temporary = SET;
    }
    else if(parentQN.t.size == sizeof(TPM_HANDLE))
    {
        // Otherwise, if the size of parent's qualified name is the size of a
        // handle, this object is a primary object
        object->attributes.primary = SET;
    }
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            object->attributes.ppsHierarchy = SET;
            break;
        case TPM_RH_OWNER:
            object->attributes.spsHierarchy = SET;
            break;
        case TPM_RH_ENDORSEMENT:
            object->attributes.epsHierarchy = SET;
```



```
            break;
        case TPM_RH_NULL:
            break;
        default:
            pAssert(FALSE);
            break;
    }
    return TPM_RC_SUCCESS;

ErrorExit:
    ObjectFlush(*handle);
    return result;
}
```

9.5.3.13 AllocateSequenceSlot()

This function allocates a sequence slot and initializes the parts that are used by the normal objects so that a sequence object is not inadvertently used for an operation that is not appropriate for a sequence.

```
static BOOL
AllocateSequenceSlot(
    TPM_HANDLE      *newHandle,     // OUT: receives the allocated handle
    HASH_OBJECT     **object,       // OUT: receives pointer to allocated object
    TPM2B_AUTH      *auth           // IN: the authValue for the slot
    )
{
    OBJECT              *objectHash;        // the hash as an object

    if(!ObjectAllocateSlot(newHandle, &objectHash))
        return FALSE;

    *object = (HASH_OBJECT *)objectHash;

    // Validate that the proper location of the hash state data relative to the
    // object state data.
    pAssert(&((*object)->auth) == &objectHash->publicArea.authPolicy);

    // Set the common values that a sequence object shares with an ordinary object
    // The type is TPM_ALG_NULL
    (*object)->type = TPM_ALG_NULL;

    // This has no name algorithm and the name is the Empty Buffer
    (*object)->nameAlg = TPM_ALG_NULL;

    // Clear the attributes
    MemorySet(&((*object)->objectAttributes), 0, sizeof(TPMA_OBJECT));

    // A sequence object is DA exempt.
    (*object)->objectAttributes.noDA = SET;

    if(auth != NULL)
    {
        MemoryRemoveTrailingZeros(auth);
        (*object)->auth = *auth;
    }
    else
        (*object)->auth.t.size = 0;
    return TRUE;
}
```

9.5.3.14 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

Table 53

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
TPM_RC
ObjectCreateHMACSequence(
    TPMI_ALG_HASH     hashAlg,         // IN: hash algorithm
    TPM_HANDLE        handle,          // IN: the handle associated with sequence
                                       //     object
    TPM2B_AUTH       *auth,            // IN: authValue
    TPMI_DH_OBJECT   *newHandle        // OUT: HMAC sequence object handle
    )
{
    HASH_OBJECT          *hmacObject;
    OBJECT               *keyObject;

    // Try to allocate a slot for new object
    if(!AllocateSequenceSlot(newHandle, &hmacObject, auth))
        return TPM_RC_OBJECT_MEMORY;

    // Set HMAC sequence bit
    hmacObject->attributes.hmacSeq = SET;

    // Get pointer to the HMAC key object
    keyObject = ObjectGet(handle);

    CryptStartHMACSequence2B(hashAlg, &keyObject->sensitive.sensitive.bits.b,
                         &hmacObject->state.hmacState);

    return TPM_RC_SUCCESS;
}
```

9.5.3.15 ObjectCreateHashSequence()

This function creates a hash sequence object.

Table 54

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
TPM_RC
ObjectCreateHashSequence(
    TPMI_ALG_HASH     hashAlg,         // IN: hash algorithm
    TPM2B_AUTH       *auth,            // IN: authValue
    TPMI_DH_OBJECT   *newHandle        // OUT: sequence object handle
    )
{
    HASH_OBJECT          *hashObject;

    // Try to allocate a slot for new object
    if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
        return TPM_RC_OBJECT_MEMORY;

    // Set hash sequence bit
    hashObject->attributes.hashSeq = SET;

    // Start hash for hash sequence
    CryptStartHashSequence(hashAlg, &hashObject->state.hashState[0]);

```

```
    return TPM_RC_SUCCESS;
}
```

9.5.3.16 ObjectCreateEventSequence()

This function creates an event sequence object.

Table 55

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
TPM_RC
ObjectCreateEventSequence(
    TPM2B_AUTH      *auth,          // IN: authValue
    TPMI_DH_OBJECT  *newHandle      // OUT: sequence object handle
    )
{
    HASH_OBJECT         *hashObject;
    UINT32               count;
    TPM_ALG_ID           hash;

    // Try to allocate a slot for new object
    if(!AllocateSequenceSlot(newHandle, &hashObject, auth))
        return TPM_RC_OBJECT_MEMORY;

    // Set the event sequence attribute
    hashObject->attributes.eventSeq = SET;


    // Initialize hash states for each implemented PCR algorithms
    for(count = 0; (hash = CryptGetHashAlgByIndex(count)) != TPM_ALG_NULL; count++)
    {
        // If this is a _TPM_Init or _TPM_HashStart, the sequence object will
        // not leave the TPM so it doesn't need the sequence handling
        if(auth == NULL)
            CryptStartHash(hash, &hashObject->state.hashState[count]);
        else
            CryptStartHashSequence(hash, &hashObject->state.hashState[count]);
    }
    return TPM_RC_SUCCESS;
}
```

9.5.3.17 ObjectTerminateEvent()

This function is called to close out the event sequence and clean up the hash context states.

```
void
ObjectTerminateEvent(
    void
    )
{
    HASH_OBJECT         *hashObject;
    int                  count;
    BYTE                 buffer[MAX_DIGEST_SIZE];
    hashObject = (HASH_OBJECT *)ObjectGet(g_DRTMHandle);

    // Don't assume that this is a proper sequence object
    if(hashObject->attributes.eventSeq)
    {
        // If it is, close any open hash contexts. This is done in case
```

```
        // the crypto implementation has some context values that need to be
        // cleaned up (hygiene).
        //
        for(count = 0; CryptGetHashAlgByIndex(count) != TPM_ALG_NULL; count++)
        {
            CryptCompleteHash(&hashObject->state.hashState[count], 0, buffer);
        }
        // Flush sequence object
        ObjectFlush(g_DRTMHandle);
    }

    g_DRTMHandle = TPM_RH_UNASSIGNED;
}
```

9.5.3.18 ObjectContextLoad()

This function loads an object from a saved object context.

Table 56

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```
TPM_RC
ObjectContextLoad(
    OBJECT          *object,        // IN: object structure from saved context
    TPMI_DH_OBJECT  *handle         // OUT: object handle
    )
{
    OBJECT      *newObject;

    // Try to allocate a slot for new object
    if(!ObjectAllocateSlot(handle, &newObject))
        return TPM_RC_OBJECT_MEMORY;

    // Copy input object data to internal structure
    *newObject = *object;

    return TPM_RC_SUCCESS;
}
```

9.5.3.19 ObjectFlush()

This function frees an object slot.

This function requires that the object is loaded.

```
void
ObjectFlush(
    TPMI_DH_OBJECT   handle         // IN: handle to be freed
    )
{
    UINT32      index = handle - TRANSIENT_FIRST;
    pAssert(ObjectIsPresent(handle));

    // Mark the handle slot as unoccupied
    s_objects[index].occupied = FALSE;

    // With no attributes
    MemorySet((BYTE*)&(s_objects[index].object.entity.attributes),
               0, sizeof(OBJECT_ATTRIBUTES));
```



```
    return;
}
```

9.5.3.20 ObjectFlushHierarchy()

This function is called to flush all the loaded transient objects associated with a hierarchy when the hierarchy is disabled.

```
void
ObjectFlushHierarchy(
    TPMI_RH_HIERARCHY    hierarchy      // IN: hierarchy to be flush
    )
{
    UINT16          i;

    // iterate object slots
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].occupied)           // If found an occupied slot
        {
            switch(hierarchy)
            {
                case TPM_RH_PLATFORM:
                    if(s_objects[i].object.entity.attributes.ppsHierarchy == SET)
                        s_objects[i].occupied = FALSE;
                    break;
                case TPM_RH_OWNER:
                    if(s_objects[i].object.entity.attributes.spsHierarchy == SET)
                        s_objects[i].occupied = FALSE;
                    break;
                case TPM_RH_ENDORSEMENT:
                    if(s_objects[i].object.entity.attributes.epsHierarchy == SET)
                        s_objects[i].occupied = FALSE;
                    break;
                default:
                    pAssert(FALSE);
                    break;
            }
        }
    }

    return;

}
```


9.5.3.21 ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that *handle* is associated with a persistent object.

Table 57

Error Returns	Meaning
TPM_RC_HANDLE	the persistent object does not exist or the associated hierarchy is disabled.
TPM_RC_OBJECT_MEMORY	no object slot

```
TPM_RC
ObjectLoadEvict(
```



```
    TPM_HANDLE       *handle,        // IN:OUT: evict object handle.  If success, it
                                     // will be replace by the loaded object handle
    TPM_CC            commandCode    // IN: the command being processed
    )
{
    TPM_RC           result;
    TPM_HANDLE       evictHandle = *handle;   // Save the evict handle
    OBJECT           *object;

    // If this is an index that references a persistent object created by
    // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
    if(*handle >=  PLATFORM_PERSISTENT)
    {
        // belongs to platform
        if(g_phEnable == CLEAR)
            return TPM_RC_HANDLE;
    }
    // belongs to owner
    else if(gc.shEnable == CLEAR)
        return TPM_RC_HANDLE;

    // Try to allocate a slot for an object
    if(!ObjectAllocateSlot(handle, &object))
        return TPM_RC_OBJECT_MEMORY;

    // Copy persistent object to transient object slot.  A TPM_RC_HANDLE
    // may be returned at this point. This will mark the slot as containing
    // a transient object so that it will be flushed at the end of the
    // command
    result = NvGetEvictObject(evictHandle, object);

    // Bail out if this failed
    if(result != TPM_RC_SUCCESS)
        return result;

    // check the object to see if it is in the endorsement hierarchy
    // if it is and this is not a TPM2_EvictControl() command, indicate
    // that the hierarchy is disabled.
    // If the associated hierarchy is disabled, make it look like the
    // handle is not defined
    if(    ObjectDataGetHierarchy(object) == TPM_RH_ENDORSEMENT
        && gc.ehEnable == CLEAR
        && commandCode != TPM_CC_EvictControl
       )
        return TPM_RC_HANDLE;

    return result;
}
```

9.5.3.22 ObjectComputeName()

This function computes the Name of an object from its public area.

```
void
ObjectComputeName(
    TPMT_PUBLIC      *publicArea,    // IN: public area of an object
    TPM2B_NAME       *name           // OUT: name of the object
    )
{
    TPM2B_PUBLIC         marshalBuffer;
    BYTE                *buffer;             // auxiliary marshal buffer pointer
    HASH_STATE           hashState;          // hash state

    // if the nameAlg is NULL then there is no name.
```



```
    if(publicArea->nameAlg == TPM_ALG_NULL)
    {
        name->t.size = 0;
        return;
    }
    // Start hash stack
    name->t.size = CryptStartHash(publicArea->nameAlg, &hashState);

    // Marshal the public area into its canonical form
    buffer = marshalBuffer.b.buffer;

    marshalBuffer.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);

    // Adding public area
    CryptUpdateDigest2B(&hashState, &marshalBuffer.b);

    // Complete hash leaving room for the name algorithm
    CryptCompleteHash(&hashState, name->t.size, &name->t.name[2]);

    // set the nameAlg
    UINT16_TO_BYTE_ARRAY(publicArea->nameAlg, name->t.name);
    name->t.size += 2;
    return;
}
```

9.5.3.23 ObjectComputeQualifiedName()

This function computes the qualified name of an object.

```
void
ObjectComputeQualifiedName(
    TPM2B_NAME      *parentQN,      // IN: parent's qualified name
    TPM_ALG_ID       nameAlg,       // IN: name hash
    TPM2B_NAME      *name,          // IN: name of the object
    TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
    )
{
    HASH_STATE      hashState;  // hash state

    //      QN_A = hash_A (QN of parent || NAME_A)

    // Start hash
    qualifiedName->t.size = CryptStartHash(nameAlg, &hashState);

    // Add parent's qualified name
    CryptUpdateDigest2B(&hashState, &parentQN->b);

    // Add self name
    CryptUpdateDigest2B(&hashState, &name->b);

    // Complete hash leaving room for the name algorithm
    CryptCompleteHash(&hashState, qualifiedName->t.size,
                      &qualifiedName->t.name[2]);
    UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
    qualifiedName->t.size += 2;
    return;
}
```

9.5.3.24 ObjectDataIsStorage()

This function determines if a public area has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

Table 58

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```
BOOL
ObjectDataIsStorage(
    TPMT_PUBLIC     *publicArea     // IN: public area of the object
    )
{
    if(   CryptIsAsymAlgorithm(publicArea->type)            // must be asymmetric,
       && publicArea->objectAttributes.restricted == SET    // restricted,
       && publicArea->objectAttributes.decrypt == SET       // decryption key
       && publicArea->objectAttributes.sign == CLEAR        // can not be sign key
      )
        return TRUE;
    else
        return FALSE;
}
```

9.5.3.25 ObjectIsStorage()

This function determines if an object has the attributes associated with a storage key. A storage key is an asymmetric object that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

Table 59

Return Value	Meaning
TRUE	if the object is a storage key
FALSE	if the object is not a storage key

```
BOOL
ObjectIsStorage(
    TPMI_DH_OBJECT   handle         // IN: object handle
    )
{
    OBJECT           *object = ObjectGet(handle);
    return ObjectDataIsStorage(&object->publicArea);
}
```

9.5.3.26 ObjectCapGetLoaded()

This function returns a a list of handles of loaded object, starting from *handle*. *Handle* must be in the range of valid transient object handles, but does not have to be the handle of a loaded transient object.

Table 60

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
ObjectCapGetLoaded(
    TPMI_DH_OBJECT   handle,        // IN: start handle
```



```
    UINT32          count,          // IN: count of returned handles
    TPML_HANDLE    *handleList      // OUT: list of handle
    )
{
    TPMI_YES_NO         more = NO;
    UINT32              i;

    pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate object slots to get loaded object handles
    for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].occupied == TRUE)
        {
            // A valid transient object can not be the copy of a persistent object
            pAssert(s_objects[i].object.entity.attributes.evict == CLEAR);

            if(handleList->count < count)
            {
                // If we have not filled up the return list, add this object
                // handle to it
                handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
                handleList->count++;
            }
            else
            {
                // If the return list is full but we still have loaded object
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }

    return more;
}
```

9.5.3.27 ObjectCapGetTransientAvail()

This function returns an estimate of the number of additional transient objects that could be loaded into the TPM.

```
UINT32
ObjectCapGetTransientAvail(
    void
    )
{
    UINT32      i;
    UINT32      num = 0;

    // Iterate object slot to get the number of unoccupied slots
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].occupied == FALSE) num++;
    }

    return num;
}
```

9.6 PCR.c

9.6.1 Introduction

This function contains the functions needed for PCR access and manipulation.

This implementation uses a static allocation for the PCR. The amount of memory is allocated based on the number of PCR in the implementation and the number of implemented hash algorithms. This is not the expected implementation. PCR SPACE DEFINITIONS.

In the definitions below, the *g_hashPcrMap* is a bit array that indicates which of the PCR are implemented. The *g_hashPcr* array is an array of digests. In this implementation, the space is allocated whether the PCR is implemented or not.

9.6.2 Includes, Defines, and Data Definitions

```
#define PCR_C
#include "InternalRoutines.h"
#include <Platform.h>
```

The initial value of PCR attributes. The value of these fields should be consistent with platform specific specifications. In this implementation, we assume the total number of implemented PCR is 24.

```
static const PCR_Attributes s_initAttributes[] =
{
    // PCR 0 - 15, static RTM
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
    {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},

    {0, 0x0F, 0x1F},        // PCR 16, Debug
    {0, 0x10, 0x1C},        // PCR 17, Locality 4
    {0, 0x10, 0x1C},        // PCR 18, Locality 3
    {0, 0x10, 0x0C},        // PCR 19, Locality 2
    {0, 0x14, 0x0E},        // PCR 20, Locality 1
    {0, 0x14, 0x04},        // PCR 21, Dynamic OS
    {0, 0x14, 0x04},        // PCR 22, Dynamic OS
    {0, 0x0F, 0x1F},        // PCR 23, App specific
    {0, 0x0F, 0x1F}         // PCR 24, testing policy
};
```

9.6.3 Functions

9.6.3.1 PCRBelongsAuthGroup()

This function indicates if a PCR belongs to a group that requires an *authValue* in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Table 61

Return Value	Meaning
TRUE:	PCR belongs an auth group
FALSE:	PCR does not belong an auth group

```
BOOL
```

```
PCRBelongsAuthGroup(
    TPMI_DH_PCR      handle,         // IN: handle of PCR
    UINT32          *groupIndex      // OUT: group index if PCR belongs a
                                     //      group that allows authValue.  If PCR
                                     //      does not belong to an auth group,
                                     //      the value in this parameter is
                                     //      invalid
)
{
#if NUM_AUTHVALUE_PCR_GROUP > 0
    // Platform specification determines to which auth group a PCR belongs (if
    // any). In this implementation, we assume there is only
    // one auth group which contains PCR[20-22].  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }

#endif
    return FALSE;
}
```

9.6.3.2 PCRBelongsPolicyGroup()

This function indicates if a PCR belongs to a group that requires a policy authorization in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Table 62

Return Value	Meaning
TRUE:	PCR belongs a policy group
FALSE:	PCR does not belong a policy group

```
BOOL
PCRBelongsPolicyGroup(
    TPMI_DH_PCR      handle,         // IN: handle of PCR
    UINT32          *groupIndex      // OUT: group index if PCR belongs a group that
                                     //     allows policy.  If PCR does not belong to
                                     //     a policy group, the value in this
                                     //     parameter is invalid
    )
{
#if NUM_POLICY_PCR_GROUP > 0
    // Platform specification decides if a PCR belongs to a policy group and
    // belongs to which group.  In this implementation, we assume there is only
    // one policy group which contains PCR20-22.  If the platform specification
    // requires differently, the implementation should be changed accordingly
    if(handle >= 20 && handle <= 22)
    {
        *groupIndex = 0;
        return TRUE;
    }
#endif
    return FALSE;
}
```

9.6.3.3 PCRBelongsTCBGroup()

This function indicates if a PCR belongs to the TCB group.

Table 63

Return Value	Meaning
TRUE:	PCR belongs to TCB group
FALSE:	PCR does not belong to TCB group

```
static BOOL
PCRBelongsTCBGroup(
    TPMI_DH_PCR      handle          // IN: handle of PCR
    )
{
#if ENABLE_PCR_NO_INCREMENT == YES
    // Platform specification decides if a PCR belongs to a TCB group.  In this
    // implementation, we assume PCR[20-22] belong to TCB group.  If the platform
    // specification requires differently, the implementation should be
    // changed accordingly
    if(handle >= 20 && handle <= 22)
        return TRUE;

#endif
    return FALSE;
}
```


9.6.3.4 PCRPolicyIsAvailable()

This function indicates if a policy is available for a PCR.

Table 64

Return Value	Meaning
TRUE	the PCR should be authorized by policy
FALSE	the PCR does not allow policy

```
BOOL
PCRPolicyIsAvailable(
    TPMI_DH_PCR      handle          // IN: PCR handle
    )
{
    UINT32          groupIndex;

    return PCRBelongsPolicyGroup(handle, &groupIndex);
}
```

9.6.3.5 PCRGetAuthValue()

This function is used to access the *authValue* of a PCR. If PCR does not belong to an *authValue* group, an Empty Auth will be returned.

```
void
PCRGetAuthValue(
    TPMI_DH_PCR      handle,         // IN: PCR handle
    TPM2B_AUTH      *auth            // OUT: authValue of PCR
    )
```



```
{
    UINT32      groupIndex;

    if(PCRBelongsAuthGroup(handle, &groupIndex))
    {
        *auth = gc.pcrAuthValues.auth[groupIndex];
    }
    else
    {
        auth->t.size = 0;
    }

    return;
}
```

9.6.3.6 PCRGetAuthPolicy()

This function is used to access the authorization policy of a PCR. It sets *policy* to the authorization policy and returns the hash algorithm for policy If the PCR does not allow a policy, TPM_ALG_NULL is returned.

```
TPMI_ALG_HASH
PCRGetAuthPolicy(
    TPMI_DH_PCR      handle,        // IN: PCR handle
    TPM2B_DIGEST    *policy         // OUT: policy of PCR
    )
{
    UINT32           groupIndex;

    if(PCRBelongsPolicyGroup(handle, &groupIndex))
    {
        *policy = gp.pcrPolicies.policy[groupIndex];
        return gp.pcrPolicies.hashAlg[groupIndex];
    }
    else
    {
        policy->t.size = 0;
        return TPM_ALG_NULL;
    }
}
```

9.6.3.7 PCRSimStart()

This function is used to initialize the policies when a TPM is manufactured. This function would only be called in a manufacturing environment or in a TPM simulator.

```
void
PCRSimStart(
    void
    )
{
    UINT32  i;
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }

    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gc.pcrAuthValues.auth[i].t.size = 0;
    }

```

```
    // We need to give an initial configuration on allocated PCR before
    // receiving any TPM2_PCR_Allocate command to change this configuration
    // When the simulation environment starts, we allocate all the PCRs
    for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
            gp.pcrAllocated.count++)
    {
        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
            = CryptGetHashAlgByIndex(gp.pcrAllocated.count);

        gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect
            = PCR_SELECT_MAX;
        for(i = 0; i < PCR_SELECT_MAX; i++)
            gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
                = 0xFF;
    }

    // Store the initial configuration to NV
    NvWriteReserved(NV_PCR_POLICIES, &gp.pcrPolicies);
    NvWriteReserved(NV_PCR_ALLOCATED, &gp.pcrAllocated);

    return;
}
```

9.6.3.8 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

Table 65

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```
static BYTE *
GetSavedPcrPointer (
    TPM_ALG_ID      alg,            // IN: algorithm for bank
    UINT32          pcrIndex        // IN: PCR index in PCR_SAVE
    )
{
    switch(alg)
    {
#ifdef TPM_ALG_SHA1
    case TPM_ALG_SHA1:
        return gc.pcrSave.sha1[pcrIndex];
        break;
#endif
#ifdef TPM_ALG_SHA256
    case TPM_ALG_SHA256:
        return gc.pcrSave.sha256[pcrIndex];
        break;
#endif
#ifdef TPM_ALG_SHA384
    case TPM_ALG_SHA384:
        return gc.pcrSave.sha384[pcrIndex];
        break;
#endif

#ifdef TPM_ALG_SHA512
    case TPM_ALG_SHA512:
        return gc.pcrSave.sha512[pcrIndex];
        break;
#endif
```



```
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        return gc.pcrSave.sm3_256[pcrIndex];
        break;
#endif
    default:
        FAIL(FATAL_ERROR_INTERNAL);
    }
    //return NULL; // Can't be reached
}
```

9.6.3.9 PcrIsAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

Table 66

Return Value	Meaning
FALSE	PCR is not allocated
TRUE	PCR is allocated

```
BOOL
PcrIsAllocated (
    UINT32          pcr,            // IN: The number of the PCR
    TPMI_ALG_HASH   hashAlg         // IN: The PCR algorithm
    )
{
    UINT32              i;
    BOOL                allocated = FALSE;

    if(pcr < IMPLEMENTATION_PCR)
    {

        for(i = 0; i < gp.pcrAllocated.count; i++)
        {
            if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
            {
                if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr/8])
                        & (1 << (pcr % 8))) != 0)
                    allocated = TRUE;
                else
                    allocated = FALSE;
                break;
            }
        }
    }
    return allocated;
}
```

9.6.3.10 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

Table 67

Return Value	Meaning
NULL	no such algorithm
not NULL	pointer to the 0th byte of the 0th PCR

```
static BYTE *
GetPcrPointer (
    TPM_ALG_ID       alg,           // IN: algorithm for bank
    UINT32           pcrNumber      // IN: PCR number
    )
{
    static BYTE     *pcr = NULL;

    if(!PcrIsAllocated(pcrNumber, alg))
        return NULL;

    switch(alg)
    {
#ifdef TPM_ALG_SHA1
    case TPM_ALG_SHA1:
        pcr = s_pcrs[pcrNumber].sha1Pcr;
        break;
#endif
#ifdef TPM_ALG_SHA256
    case TPM_ALG_SHA256:
        pcr = s_pcrs[pcrNumber].sha256Pcr;
        break;
#endif
#ifdef TPM_ALG_SHA384
    case TPM_ALG_SHA384:
        pcr = s_pcrs[pcrNumber].sha384Pcr;
        break;
#endif
#ifdef TPM_ALG_SHA512
    case TPM_ALG_SHA512:
        pcr = s_pcrs[pcrNumber].sha512Pcr;
        break;
#endif
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        pcr = s_pcrs[pcrNumber].sm3_256Pcr;
        break;
#endif
    default:
        pAssert(FALSE);
        break;
    }

    return pcr;
}
```

9.6.3.11 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in *selection*.

Table 68

Return Value	Meaning
FALSE	PCR is not selected
TRUE	PCR is selected

```
static BOOL
IsPcrSelected (
    UINT32                pcr,           // IN: The number of the PCR
    TPMS_PCR_SELECTION   *selection      // IN: The selection structure
    )
{
    BOOL                  selected = FALSE;
    if(   pcr < IMPLEMENTATION_PCR
       && ((selection->pcrSelect[pcr/8]) & (1 << (pcr % 8))) != 0)
        selected = TRUE;

    return selected;
}
```

9.6.3.12 FilterPcr()

This function modifies a PCR selection array based on the implemented PCR.

```
static void
FilterPcr(
    TPMS_PCR_SELECTION   *selection      // IN: input PCR selection
    )
{
    UINT32     i;
    TPMS_PCR_SELECTION      *allocated = NULL;

    // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
    for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
        selection->pcrSelect[i] = 0;

    // Find the internal configuration for the bank
    for(i = 0; i < gp.pcrAllocated.count; i++)
    {
        if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
        {
            allocated = &gp.pcrAllocated.pcrSelections[i];
            break;
        }
    }

    for (i = 0; i < selection->sizeofSelect; i++)
    {
        if(allocated == NULL)
        {
            // If the required bank does not exist, clear input selection
            selection->pcrSelect[i] = 0;
        }
        else
            selection->pcrSelect[i] &= allocated->pcrSelect[i];
    }

    return;
}
```

9.6.3.13 PcrDrtm()

This function does the DRTM and H-CRTM processing it is called from _TPM_Hash_End().

```
void
PcrDrtm(
    const TPMI_DH_PCR        pcrHandle,     // IN: the index of the PCR to be
                                            //     modified
    const TPMI_ALG_HASH      hash,          // IN: the bank identifier
    const TPM2B_DIGEST      *digest         // IN: the digest to modify the PCR
    )
{
    BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);

    if(pcrData != NULL)
    {
        // Rest the PCR to zeros
        MemorySet(pcrData, 0, digest->t.size);

        // if the TPM has not started, then set the PCR to 0...04 and then extend
        if(!TPMIsStarted())
        {
            pcrData[digest->t.size - 1] = 4;
        }
        // Now, extend the value
        PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
    }
}
```

9.6.3.14 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```
void
PCRStartup(
    STARTUP_TYPE     type            // IN: startup type
    )
{
    UINT32              pcr, j;
    UINT32              saveIndex = 0;

    g_pcrReConfig = FALSE;

    if(type != SU_RESUME)
    {
        // PCR generation counter is cleared at TPM_RESET and TPM_RESTART
        gr.pcrCounter = 0;
    }

    // Initialize/Restore PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        BOOL        incrSaveIndex = FALSE;

        // If this is the H-CRTM PCR and we are not doing a resume and we
        // had an H-CRTM event, then we don't change this PCR
        if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
            continue;

        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
            BYTE            *pcrData = GetPcrPointer(hash, pcr);
```



```
            UINT16          pcrSize = CryptGetHashDigestSize(hash);

            if(pcrData != NULL)
            {
                if(type == SU_RESUME && s_initAttributes[pcr].stateSave == SET)
                {
                    // Restore saved PCR value
                    BYTE    *pcrSavedData;
                    pcrSavedData = GetSavedPcrPointer(
                                       gp.pcrAllocated.pcrSelections[j].hash,
                                       saveIndex);
                    MemoryCopy(pcrData, pcrSavedData, pcrSize, pcrSize);
                    incrSaveIndex = TRUE;
                }
                else
                    // PCR was not restored by state save
                {
                    // If the reset locality of the PCR is 4, then
                    // the reset value is all one's, otherwise it is
                    // all zero.
                    if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                        MemorySet(pcrData, 0xFF, pcrSize);
                    else
                        MemorySet(pcrData, 0, pcrSize);
                }
            }
        }
        if(incrSaveIndex == TRUE)
            saveIndex++;
    }

    // Reset authValues
    if(type != SU_RESUME)
    {
        for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
        {
            gc.pcrAuthValues.auth[j].t.size = 0;
        }
    }

}
```

9.6.3.15 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

```
void
PCRStateSave(
    TPM_SU          type            // IN: startup type
    )
{
    UINT32          pcr, j;
    UINT32          saveIndex = 0;

    // if state save CLEAR, nothing to be done.  Return here
    if(type == TPM_SU_CLEAR) return;

    // Copy PCR values to the structure that should be saved to NV
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            BYTE    *pcrData;
```

```
                UINT32   pcrSize;

                pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);

                if(pcrData != NULL)
                {
                    pcrSize
                        = CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[j].hash);

                    if(s_initAttributes[pcr].stateSave == SET)
                    {
                        // Restore saved PCR value
                        BYTE     *pcrSavedData;
                        pcrSavedData
                            = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
                                                 saveIndex++);
                        MemoryCopy(pcrSavedData, pcrData, pcrSize, pcrSize);
                    }
                }
            }
        }

        return;
}
```

9.6.3.16 PCRIsStateSaved()

This function indicates if the selected PCR is a PCR that is state saved on TPM2_Shutdown(STATE). The return value is based on PCR attributes.

Table 69

Return Value	Meaning
TRUE	PCR is state saved
FALSE	PCR is not state saved

```
BOOL
PCRIsStateSaved(
    TPMI_DH_PCR      handle           // IN: PCR handle to be extended
    )
{
    UINT32              pcr = handle - PCR_FIRST;

    if(s_initAttributes[pcr].stateSave == SET)
        return TRUE;
    else
        return FALSE;
}
```


9.6.3.17 PCRIsResetAllowed()

This function indicates if a PCR may be reset by the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Table 70

Return Value	Meaning
TRUE	TPM2_PCR_Reset() is allowed
FALSE	TPM2_PCR_Reset() is not allowed

```
BOOL
PCRIsResetAllowed(
    TPMI_DH_PCR      handle         // IN: PCR handle to be extended
    )
{
    UINT8               commandLocality;
    UINT8               localityBits = 1;
    UINT32              pcr = handle - PCR_FIRST;

    // Check for the locality
    commandLocality = _plat__LocalityGet();

#ifdef DRTM_PCR
    // For a TPM that does DRTM, Reset is not allowed at locality 4
    if(commandLocality == 4)
        return FALSE;
#endif

    localityBits = localityBits << commandLocality;
    if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
        return FALSE;
    else
        return TRUE;

}
```

9.6.3.18 PCRChanged()

This function checks a PCR handle to see if the attributes for the PCR are set so that any change to the PCR causes an increment of the *pcrCounter*. If it does, then the function increments the counter.

```
void
PCRChanged(
    TPM_HANDLE       pcrHandle      // IN: the handle of the PCR that changed.
    )
{
    // For the reference implementation, the only change that does not cause
    // increment is a change to a PCR in the TCB group.
    if(!PCRBelongsTCBGroup(pcrHandle))
        gr.pcrCounter++;
}
```

9.6.3.19 PCRIsExtendAllowed()

This function indicates a PCR may be extended at the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Table 71

Return Value	Meaning
TRUE	extend is allowed
FALSE	extend is not allowed

```
BOOL
PCRIsExtendAllowed(
    TPMI_DH_PCR      handle         // IN: PCR handle to be extended
    )
{
    UINT8                commandLocality;
    UINT8                localityBits = 1;
    UINT32               pcr = handle - PCR_FIRST;

    // Check for the locality
    commandLocality = _plat__LocalityGet();
    localityBits = localityBits << commandLocality;
    if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
        return FALSE;
    else
        return TRUE;

}
```

9.6.3.20 PCRExtend()

This function is used to extend a PCR in a specific bank.

```
void
PCRExtend(
    TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
    TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
    UINT32           size,          // IN: size of data to be extended
    BYTE            *data           // IN: data to be extended
    )
{
    UINT32               pcr = handle - PCR_FIRST;
    BYTE                *pcrData;
    HASH_STATE           hashState;
    UINT16               pcrSize;

    pcrData = GetPcrPointer(hash, pcr);

    // Extend PCR if it is allocated
    if(pcrData != NULL)
    {
        pcrSize = CryptGetHashDigestSize(hash);
        CryptStartHash(hash, &hashState);
        CryptUpdateDigest(&hashState, pcrSize, pcrData);
        CryptUpdateDigest(&hashState, size, data);
        CryptCompleteHash(&hashState, pcrSize, pcrData);

        // If PCR does not belong to TCB group, increment PCR counter
        if(!PCRBelongsTCBGroup(handle))
            gr.pcrCounter++;
    }

    return;
}
```

9.6.3.21 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

As a side-effect, *selection* is modified so that only the implemented PCR will have their bits still set.

```
void
PCRComputeCurrentDigest(
    TPMI_ALG_HASH        hashAlg,       // IN: hash algorithm to compute digest
    TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
                                        //     output)
    TPM2B_DIGEST        *digest         // OUT: digest
    )
{
    HASH_STATE                 hashState;
    TPMS_PCR_SELECTION        *select;
    BYTE                      *pcrData;   // will point to a digest
    UINT32                     pcrSize;
    UINT32                     pcr;
    UINT32                     i;

    // Initialize the hash
    digest->t.size = CryptStartHash(hashAlg, &hashState);
    pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);

    // Iterate through the list of PCR selection structures
    for(i = 0; i < selection->count; i++)
    {
        // Point to the current selection
        select = &selection->pcrSelections[i]; // Point to the current selection
        FilterPcr(select);      // Clear out the bits for unimplemented PCR

        // Need the size of each digest
        pcrSize = CryptGetHashDigestSize(selection->pcrSelections[i].hash);

        // Iterate through the selection
        for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            if(IsPcrSelected(pcr, select))         // Is this PCR selected
            {
                // Get pointer to the digest data for the bank
                pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
                pAssert(pcrData != NULL);
                CryptUpdateDigest(&hashState, pcrSize, pcrData);  // add to digest
            }
        }
    }
    // Complete hash stack
    CryptCompleteHash2B(&hashState, &digest->b);

    return;
}
```


9.6.3.22 PCRRead()

This function is used to read a list of selected PCR. If the requested PCR number exceeds the maximum number that can be output, the *selection* is adjusted to reflect the actual output PCR.

```
void
PCRRead(
    TPML_PCR_SELECTION  *selection,     // IN/OUT: PCR selection (filtered on
                                        //     output)
    TPML_DIGEST         *digest,        // OUT: digest
    UINT32              *pcrCounter     // OUT: the current value of PCR generation
```



```
                                        //     number
    )
{
    TPMS_PCR_SELECTION      *select;
    BYTE                    *pcrData;       // will point to a digest
    UINT32                   pcr;
    UINT32                   i;

    digest->count = 0;

    // Iterate through the list of PCR selection structures
    for(i = 0; i < selection->count; i++)
    {
        // Point to the current selection
        select = &selection->pcrSelections[i]; // Point to the current selection
        FilterPcr(select);      // Clear out the bits for unimplemented PCR

        // Iterate through the selection
        for (pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
        {
            if(IsPcrSelected(pcr, select))         // Is this PCR selected
            {
                // Check if number of digest exceed upper bound
                if(digest->count > 7)
                {
                    // Clear rest of the current select bitmap
                    while(   pcr < IMPLEMENTATION_PCR
                             // do not round up!
                          && (pcr / 8) < select->sizeofSelect)
                    {
                        // do not round up!
                        select->pcrSelect[pcr/8] &= (BYTE) ~(1 << (pcr % 8));
                        pcr++;
                    }
                    // Exit inner loop
                    break;;
                }
                // Need the size of each digest
                digest->digests[digest->count].t.size =
                    CryptGetHashDigestSize(selection->pcrSelections[i].hash);

                // Get pointer to the digest data for the bank
                pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
                pAssert(pcrData != NULL);
                // Add to the data to digest
                MemoryCopy(digest->digests[digest->count].t.buffer,
                           pcrData,
                           digest->digests[digest->count].t.size,
                           digest->digests[digest->count].t.size);
                digest->count++;
            }
        }
        // If we exit inner loop because we have exceed the output upper bound
        if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
        {
            // Clear rest of the selection
            while(i < selection->count)
            {
                MemorySet(selection->pcrSelections[i].pcrSelect, 0,
                          selection->pcrSelections[i].sizeofSelect);
                i++;
            }
            // exit outer loop
            break;
        }
    }
```

```
    *pcrCounter = gr.pcrCounter;

    return;
}
```

9.6.3.23 PcrWrite()

This function is used by _TPM_Hash_End() to set a PCR to the computed hash of the H-CRTM event.

```
void
PcrWrite(
    TPMI_DH_PCR      handle,        // IN: PCR handle to be extended
    TPMI_ALG_HASH    hash,          // IN: hash algorithm of PCR
    TPM2B_DIGEST    *digest         // IN: the new value
    )
{
    UINT32               pcr = handle - PCR_FIRST;
    BYTE                *pcrData;

    // Copy value to the PCR if it is allocated
    pcrData = GetPcrPointer(hash, pcr);
    if(pcrData != NULL)
    {
        MemoryCopy(pcrData, digest->t.buffer, digest->t.size, digest->t.size); ;
    }

    return;
}
```

9.6.3.24 PCRAllocate()

This function is used to change the PCR allocation.

Table 72

Error Returns	Meaning
TPM_RC_SUCCESS	allocate success
TPM_RC_NO_RESULTS	allocate failed
TPM_RC_PCR	improper allocation

```
TPM_RC
PCRAllocate(
    TPML_PCR_SELECTION  *allocate,      // IN: required allocation
    UINT32              *maxPCR,        // OUT: Maximum number of PCR
    UINT32              *sizeNeeded,    // OUT: required space
    UINT32              *sizeAvailable  // OUT: available space
    )
{
    UINT32                  i, j, k;
    TPML_PCR_SELECTION      newAllocate;
    // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
    BOOL                    pcrHcrtm = FALSE;
    BOOL                    pcrDrtm = FALSE;

    // Create the expected new PCR allocation based on the existing allocation
    // and the new input:
    //  1. if a PCR bank does not appear in the new allocation, the existing
    //     allocation of this PCR bank will be preserved.
    //  2. if a PCR bank appears multiple times in the new allocation, only the
```

```
    //     last one will be in effect.
    newAllocate = gp.pcrAllocated;
    for(i = 0; i < allocate->count; i++)
    {
        for(j = 0; j < newAllocate.count; j++)
        {
            // If hash matches, the new allocation covers the old allocation
            // for this particular bank.
            // The assumption is the initial PCR allocation (from manufacture)
            // has all the supported hash algorithms with an assigned bank
            // (possibly empty).  So there must be a match for any new bank
            // allocation from the input.
            if(newAllocate.pcrSelections[j].hash ==
                allocate->pcrSelections[i].hash)
            {
                newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
                break;
            }
        }
        // The j loop must exit with a match.
        pAssert(j < newAllocate.count);
    }

    // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
    *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
    if(*maxPCR > IMPLEMENTATION_PCR)
        *maxPCR = IMPLEMENTATION_PCR;

    // Compute required size for allocation
    *sizeNeeded = 0;
    for(i = 0; i < newAllocate.count; i++)
    {
        UINT32      digestSize
            = CryptGetHashDigestSize(newAllocate.pcrSelections[i].hash);
#if defined(DRTM_PCR)
        // Make sure that we end up with at least one DRTM PCR
#   define PCR_DRTM  (PCR_FIRST + DRTM_PCR)    // for cosmetics
        pcrDrtm =    pcrDrtm || TEST_BIT(PCR_DRTM, newAllocate.pcrSelections[i]);
#else   // if DRTM PCR is not required, indicate that the allocation is OK
        pcrDrtm = TRUE;
#endif

#if defined(HCRTM_PCR)
        // and one HCRTM PCR (since this is usually PCR 0...)
#   define PCR_HCRTM (PCR_FIRST + HCRTM_PCR)
        pcrHcrtm =    pcrDrtm || TEST_BIT(PCR_HCRTM, newAllocate.pcrSelections[i]);
#else
        pcrHcrtm = TRUE;
#endif
        for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
        {
            BYTE        mask = 1;
            for(k = 0; k < 8; k++)
            {
                if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
                    *sizeNeeded += digestSize;
                mask = mask << 1;
            }
        }
    }

    if(!pcrDrtm || !pcrHcrtm)
        return TPM_RC_PCR;


    // In this particular implementation, we always have enough space to
```



```
    // allocate PCR.  Different implementation may return a sizeAvailable less
    // than the sizeNeed.
    *sizeAvailable = sizeof(s_pcrs);

    // Save the required allocation to NV.  Note that after NV is written, the
    // PCR allocation in NV is no longer consistent with the RAM data
    // gp.pcrAllocated.  The NV version reflect the allocate after next
    // TPM_RESET, while the RAM version reflects the current allocation
    NvWriteReserved(NV_PCR_ALLOCATED, &newAllocate);

    return TPM_RC_SUCCESS;

}
```

9.6.3.25 PCRSetValue()

This function is used to set the designated PCR in all banks to an initial value. The initial value is signed and will be sign extended into the entire PCR.

```
void
PCRSetValue(
    TPM_HANDLE       handle,        // IN: the handle of the PCR to set
    INT8             initialValue   // IN: the value to set
    )
{
    int              i;
    UINT32           pcr = handle - PCR_FIRST;
    TPMI_ALG_HASH    hash;
    UINT16           digestSize;
    BYTE            *pcrData;

    // Iterate supported PCR bank algorithms to reset
    for(i = 0; i < HASH_COUNT; i++)
    {
        hash = CryptGetHashAlgByIndex(i);
        // Prevent runaway
        if(hash == TPM_ALG_NULL)
            break;

        // Get a pointer to the data
        pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);

        // If the PCR is allocated
        if(pcrData != NULL)
        {
            // And the size of the digest
            digestSize = CryptGetHashDigestSize(hash);

            // Set the LSO to the input value
            pcrData[digestSize - 1] = initialValue;

            // Sign extend
            if(initialValue >= 0)
                MemorySet(pcrData, 0, digestSize - 1);
            else
                MemorySet(pcrData, -1, digestSize - 1);
        }
    }
}
```

9.6.3.26 PCRResetDynamics

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

```
void
PCRResetDynamics(
    void
    )
{
    UINT32              pcr, i;

    // Initialize PCR values
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        // Iterate each hash algorithm bank
        for(i = 0; i < gp.pcrAllocated.count; i++)
        {
            BYTE    *pcrData;
            UINT32  pcrSize;

            pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);

            if(pcrData != NULL)
            {
                pcrSize =
                    CryptGetHashDigestSize(gp.pcrAllocated.pcrSelections[i].hash);

                // Reset PCR
                // Any PCR can be reset by locality 4 should be reset to 0
                if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                    MemorySet(pcrData, 0, pcrSize);
            }
        }
    }
    return;
}
```

9.6.3.27 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Table 73

Return Value	Meaning
YES:	if the return count is 0
NO:	if the return count is not 0

```
TPMI_YES_NO
PCRCapGetAllocation(
    UINT32              count,          // IN: count of return
    TPML_PCR_SELECTION  *pcrSelection   // OUT: PCR allocation list
    )
{
    if(count == 0)
    {
        pcrSelection->count = 0;
        return YES;
    }
    else
    {
        *pcrSelection = gp.pcrAllocated;
        return NO;
    }
}
```


9.6.3.28 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```
static void
PCRSetSelectBit(
    UINT32            pcr,            // IN: PCR number
    BYTE             *bitmap          // OUT: bit map to be set
    )
{
    bitmap[pcr / 8] |= (1 << (pcr % 8));
    return;
}
```

9.6.3.29 PCRGetProperty()

This function returns the selected PCR property.

Table 74

Return Value	Meaning
TRUE	the property type is implemented
FALSE	the property type is not implemented

```
static BOOL
PCRGetProperty(
    TPM_PT_PCR                 property,
    TPMS_TAGGED_PCR_SELECT    *select
    )
{
    UINT32              pcr;
    UINT32              groupIndex;

    select->tag = property;
    // Always set the bitmap to be the size of all PCR
    select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;

    // Initialize bitmap
    MemorySet(select->pcrSelect, 0, select->sizeofSelect);

    // Collecting properties
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        switch(property)
        {
        case TPM_PT_PCR_SAVE:
            if(s_initAttributes[pcr].stateSave == SET)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L0:
            if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_RESET_L0:
            if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L1:
            if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
```

```
            case TPM_PT_PCR_RESET_L1:
                if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_EXTEND_L2:
                if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_RESET_L2:
                if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_EXTEND_L3:
                if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_RESET_L3:
                if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_EXTEND_L4:
                if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_RESET_L4:
                if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
            case TPM_PT_PCR_DRTM_RESET:
                // DRTM reset PCRs are the PCR reset by locality 4
                if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
#if NUM_POLICY_PCR_GROUP > 0
            case TPM_PT_PCR_POLICY:
                if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
#endif
#if NUM_AUTHVALUE_PCR_GROUP > 0
            case TPM_PT_PCR_AUTH:
                if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
#endif
#if ENABLE_PCR_NO_INCREMENT == YES
            case TPM_PT_PCR_NO_INCREMENT:
                if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
                    PCRSetSelectBit(pcr, select->pcrSelect);
                break;
#endif
            default:
                // If property is not supported, stop scanning PCR attributes
                // and return.
                return FALSE;
                break;
        }
    }
    return TRUE;
}
```


9.6.3.30 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

Table 75

Return Value	Meaning
YES:	if no more property is available
NO:	if there are more properties not reported

```
TPMI_YES_NO
PCRCapGetProperties(
    TPM_PT_PCR                    property,      // IN: the starting PCR property
    UINT32                        count,         // IN: count of returned properties
    TPML_TAGGED_PCR_PROPERTY     *select         // OUT: PCR select
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // Initialize output property list
    select->count = 0;

    // The maximum count of properties we may return is MAX_PCR_PROPERTIES
    if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;

    // TPM_PT_PCR_FIRST is defined as 0 in spec.  It ensures that property
    // value would never be less than TPM_PT_PCR_FIRST
    pAssert(TPM_PT_PCR_FIRST == 0);

    // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
    // implemented on the TPM.
    for(i = property; i <= TPM_PT_PCR_LAST; i++)
    {
        if(select->count < count)
        {
            // If we have not filled up the return list, add more properties to it
            if(PCRGetProperty(i, &select->pcrProperty[select->count]))
                // only increment if the property is implemented
            select->count++;
        }
        else
        {
            // If the return list is full but we still have properties
            // available, report this and stop iterating.
            more = YES;
            break;
        }
    }
    return more;
}
```


9.6.3.31 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

Table 76

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
```

```
PCRCapGetHandles(
    TPMI_DH_PCR      handle,          // IN: start handle
    UINT32           count,           // IN: count of returned handles
    TPML_HANDLE     *handleList       // OUT: list of handle
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    pAssert(HandleGetType(handle) == TPM_HT_PCR);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate PCR handle range
    for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this PCR
            // handle to it
            handleList->handle[handleList->count] = i + PCR_FIRST;
            handleList->count++;
        }
        else
        {
            // If the return list is full but we still have PCR handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

9.7 PP.c

9.7.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

9.7.2 Includes

```
#include "InternalRoutines.h"
```

9.7.3 Functions

9.7.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that require confirmation with physical presence. The array is an array of bits that has a correspondence with the command code.

This command should only ever be executable in a manufacturing setting or in a simulation.

```
void
PhysicalPresencePreInstall_Init(
    void
```



```
    )
{
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0,
              ((TPM_CC_PP_LAST - TPM_CC_PP_FIRST + 1) + 7) / 8);

    // TPM_CC_PP_Commands always requires PP
    if(CommandIsImplemented(TPM_CC_PP_Commands))
        PhysicalPresenceCommandSet(TPM_CC_PP_Commands);

    // Write PP list to NV
    NvWriteReserved(NV_PP_LIST, &gp.ppList);

    return;
}
```

9.7.3.2 PhysicalPresenceCommandSet()

This function is used to indicate a command that requires PP confirmation.

```
void
PhysicalPresenceCommandSet(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Set bit
    gp.ppList[bitPos/8] |= 1 << (bitPos % 8);

    return;
}
```

9.7.3.3 PhysicalPresenceCommandClear()

This function is used to indicate a command that no longer requires PP confirmation.

```
void
PhysicalPresenceCommandClear(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // Assume command is implemented.  It should be checked before this
    // function is called
    pAssert(CommandIsImplemented(commandCode));

    // If the command is not a PP command, ignore it
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return;

    // if the input code is TPM_CC_PP_Commands, it can not be cleared
```

```
    if(commandCode == TPM_CC_PP_Commands)
        return;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Set bit
    gp.ppList[bitPos/8] |= (1 << (bitPos % 8));
    // Flip it to off
    gp.ppList[bitPos/8] ^= (1 << (bitPos % 8));

    return;
}
```

9.7.3.4 PhysicalPresenceIsRequired()

This function indicates if PP confirmation is required for a command.

Table 77

Return Value	Meaning
TRUE	if physical presence is required
FALSE	if physical presence is not required

```
BOOL
PhysicalPresenceIsRequired(
    TPM_CC          commandCode    // IN: command code
    )
{
    UINT32      bitPos;

    // if the input commandCode is not a PP command, return FALSE
    if(commandCode < TPM_CC_PP_FIRST || commandCode > TPM_CC_PP_LAST)
        return FALSE;

    bitPos = commandCode - TPM_CC_PP_FIRST;

    // Check the bit map.  If the bit is SET, PP authorization is required
    return ((gp.ppList[bitPos/8] & (1 << (bitPos % 8))) != 0);

}
```

9.7.3.5 PhysicalPresenceCapGetCCList()

This function returns a list of commands that require PP confirmation. The list starts from the first implemented command that has a command code that the same or greater than *commandCode*.

Table 78

Return Value	Meaning
YES	if there are more command codes available
NO	all the available command codes have been returned

```
TPMI_YES_NO
PhysicalPresenceCapGetCCList(
    TPM_CC          commandCode,    // IN: start command code
    UINT32          count,          // IN: count of returned TPM_CC
    TPML_CC        *commandList     // OUT: list of TPM_CC
    )
```



```
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // Initialize output handle list
    commandList->count = 0;

    // The maximum count of command we may return is MAX_CAP_CC
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

    // Collect PP commands
    for(i = commandCode; i <= TPM_CC_PP_LAST; i++)
    {
        if(PhysicalPresenceIsRequired(i))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to it
                commandList->commandCodes[commandList->count] = i;
                commandList->count++;
            }
            else
            {
                // If the return list is full but we still have PP command
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }
    return more;
}
```

9.8 Session.c

9.8.1 Introduction

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated.

The counter *(contextCounter)*in this implementation is a UINT64 but can be smaller. The "tracking array" *(contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible.

The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional 2^16-1 contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions.

The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context).

The expected remediation by the TRM is to load the oldest saved session context (the one found by the TMP), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

In the worst case, software may have to load and save several contexts in order to save an additional one. This should happen very infrequently.

When the TPM searches *contextArray* and finds that none of the *contextIDs* match the low-order 16-bits of *contextCount*, the TPM can copy the low bits to the *contextArray* associated with the session, and increment *contextCount*.

There is one entry in *contextArray* for each of the active sessions allowed by the TPM implementation. This array contains either a context count, an index, or a value indicating the slot is available (0).

The index into the *contextArray* is the handle for the session with the region selector byte of the session set to zero. If an entry in *contextArray* contains 0, then the corresponding handle may be assigned to a session. If the entry contains a value that is less than or equal to the number of loaded sessions for the TPM, then the array entry is the slot in which the context is loaded.

EXAMPLE If the TPM allows 8 loaded sessions, then the slot numbers would be 1-8 and a *contextArrary* value in that range would represent the loaded session.

NOTE When the TPM firmware determines that the array entry is for a loaded session, it will subtract 1 to create the zero-based slot number.

There is one significant corner case in this scheme. When the *contextCount* is equal to a value in the *contextArray*, the oldest session needs to be recycled or flushed. In order to recycle the session, it must be loaded. To be loaded, there must be an available slot. Rather than require that a spare slot be available all the time, the TPM will check to see if the *contextCount* is equal to some value in the *contextArray* when a session is created. This prevents the last session slot from being used when it is likely that a session will need to be recycled.

If a TPM with both 1. 2 and 2. 0 functionality uses this scheme for both 1. 2 and 2. 0 sessions, and the list of active contexts is read with TPM_GetCapabiltiy(), the TPM will create 32-bit representations of the list that contains 16-bit values (the TPM2_GetCapability() returns a list of handles for active sessions rather than a list of *contextID*). The full *contextID* has high-order bits that are either the same as the current *contextCount* or one less. It is one less if the 16-bits of the *contextArray* has a value that is larger than the low-order 16 bits of *contextCount*.

9.8.2 Includes, Defines, and Local Variables

```
#define SESSION_C
#include "InternalRoutines.h"
#include "Platform.h"
#include "SessionProcess_fp.h"
```

9.8.3 File Scope Function -- ContextIdSetOldest()

This function is called when the oldest *contextID* is being loaded or deleted. Once a saved context becomes the oldest, it stays the oldest until it is deleted.

Finding the oldest is a bit tricky. It is not just the numeric comparison of values but is dependent on the value of *contextCounter*.

EXAMPLE Assume we have a small *contextArray* with 8, 4-bit values with values 1 and 2 used to indicate the loaded context slot number. Also assume that the array contains hex values of (0 0 1 0 3 0 9 F) and that the *contextCounter* is an 8-bit counter with a value of 0x37. Since the low nibble is 7, that means that values above 7 are older than values below it and, in this example, 9 is the oldest value.

NOTE if we subtract the counter value, from each slot that contains a saved *contextID* we get (- - - - B - 2 - 8) and the oldest entry is now easy to find.

```
static void
ContextIdSetOldest(
    void
```



```
    )
{
    CONTEXT_SLOT    lowBits;
    CONTEXT_SLOT    entry;
    CONTEXT_SLOT    smallest = ((CONTEXT_SLOT) ~0);
    UINT32  i;

    // Set oldestSaveContext to a value indicating none assigned
    s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;

    lowBits = (CONTEXT_SLOT)gr.contextCounter;
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        entry = gr.contextArray[i];

        // only look at entries that are saved contexts
        if(entry > MAX_LOADED_SESSIONS)
        {
            // Use a less than or equal in case the oldest
            // is brand new (= lowBits-1) and equal to our initial
            // value for smallest.
            if(((CONTEXT_SLOT) (entry - lowBits)) <= smallest)
            {
                smallest = (entry - lowBits);
                s_oldestSavedSession = i;
            }
        }
    }
    // When we finish, either the s_oldestSavedSession still has its initial
    // value, or it has the index of the oldest saved context.
}
```

9.8.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
void
SessionStartup(
    STARTUP_TYPE     type
    )
{
    UINT32                  i;

    // Initialize session slots.  At startup, all the in-memory session slots
    // are cleared and marked as not occupied
    for(i = 0; i < MAX_LOADED_SESSIONS; i++)
        s_sessions[i].occupied = FALSE;   // session slot is not occupied

    // The free session slots the number of maximum allowed loaded sessions
    s_freeSessionSlots = MAX_LOADED_SESSIONS;

    // Initialize context ID data.  On a ST_SAVE or hibernate sequence, it will
    // scan the saved array of session context counts, and clear any entry that
    // references a session that was in memory during the state save since that
    // memory was not preserved over the ST_SAVE.
    if(type == SU_RESUME || type == SU_RESTART)
    {
        // On ST_SAVE we preserve the contexts that were saved but not the ones
        // in memory
        for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
        {
            // If the array value is unused or references a loaded session then
            // that loaded session context is lost and the array entry is
            // reclaimed.
```

```
                if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
                    gr.contextArray[i] = 0;
            }
            // Find the oldest session in context ID data and set it in
            // s_oldestSavedSession
            ContextIdSetOldest();
        }
        else
        {
            // For STARTUP_CLEAR, clear out the contextArray
            for (i = 0; i < MAX_ACTIVE_SESSIONS; i++)
                gr.contextArray[i] = 0;

            // reset the context counter
            gr.contextCounter = MAX_LOADED_SESSIONS + 1;

            // Initialize oldest saved session
            s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
        }
        return;
}
```

9.8.5 Access Functions

9.8.5.1 SessionIsLoaded()

This function test a session handle references a loaded session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE A PWAP authorization does not have a session.

Table 79

Return Value	Meaning
TRUE	if session is loaded
FALSE	if it is not loaded

```
BOOL
SessionIsLoaded(
    TPM_HANDLE          handle          // IN: session handle
    )
{
    pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);

    handle = handle & HR_HANDLE_MASK;

    // if out of range of possible active session, or not assigned to a loaded
    // session return false
    if(   handle >= MAX_ACTIVE_SESSIONS
       || gr.contextArray[handle] == 0
       || gr.contextArray[handle] > MAX_LOADED_SESSIONS
      )
        return FALSE;

    return TRUE;
}
```


9.8.5.2 SessionIsSaved()

This function test a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE An password authorization does not have a session.

This function requires that the handle be a valid session handle.

Table 80

Return Value	Meaning
TRUE	if session is saved
FALSE	if it is not saved

```
BOOL
SessionIsSaved(
    TPM_HANDLE       handle         // IN: session handle
    )
{
    pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION);

    handle = handle & HR_HANDLE_MASK;
    // if out of range of possible active session, or not assigned, or
    // assigned to a loaded session, return false
    if(   handle >= MAX_ACTIVE_SESSIONS
       || gr.contextArray[handle] == 0
       || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
      )
        return FALSE;

    return TRUE;
}
```

9.8.5.3 SessionPCRValueIsCurrent()

This function is used to check if PCR values have been updated since the last time they were checked in a policy session.

This function requires the session is loaded.

Table 81

Return Value	Meaning
TRUE	if PCR value is current
FALSE	if PCR value is not current

```
BOOL
SessionPCRValueIsCurrent(
    TPMI_SH_POLICY   handle         // IN: session handle
    )
{
    SESSION             *session;

    pAssert(SessionIsLoaded(handle));

    session = SessionGet(handle);
```

```
    if(   session->pcrCounter != 0
       && session->pcrCounter != gr.pcrCounter
      )
        return FALSE;
    else
        return TRUE;
}
```

9.8.5.4 SessionGet()

This function returns a pointer to the session object associated with a session handle.

The function requires that the session is loaded.

```
SESSION *
SessionGet(
    TPM_HANDLE       handle         // IN: session handle
    )
{
    CONTEXT_SLOT    sessionIndex;

    pAssert(   HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION
           );

    pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);

    // get the contents of the session array.  Because session is loaded, we
    // should always get a valid sessionIndex
    sessionIndex = gr.contextArray[handle & HR_HANDLE_MASK] - 1;

    pAssert(sessionIndex < MAX_LOADED_SESSIONS);

    return &s_sessions[sessionIndex].session;
}
```

9.8.6 Utility Functions

9.8.6.1 ContextIdSessionCreate()

This function is called when a session is created. It will check to see if the current gap would prevent a context from being saved. If so it will return TPM_RC_CONTEXT_GAP. Otherwise, it will try to find an open slot in *contextArray*, set *contextArray* to the slot.

This routine requires that the caller has determined the session array index for the session.

Table 82

return type	TPM_RC
TPM_RC_SUCCESS	context ID was assigned
TPM_RC_CONTEXT_GAP	can't assign a new *contextID* until the oldest saved session context is recycled
TPM_RC_SESSION_HANDLE	there is no slot available in the context array for tracking of this session context

```
static TPM_RC
ContextIdSessionCreate (
    TPM_HANDLE      *handle,        // OUT: receives the assigned handle. This will
                                    //     be an index that must be adjusted by the
```



```
                                   //     caller according to the type of the
                                   //     session created
    UINT32            sessionIndex   // IN: The session context array entry that will
                                   //     be occupied by the created session
    )
{

    pAssert(sessionIndex < MAX_LOADED_SESSIONS);

    // check to see if creating the context is safe
    // Is this going to be an assignment for the last session context
    // array entry?  If so, then there will be no room to recycle the
    // oldest context if needed.  If the gap is not at maximum, then
    // it will be possible to save a context if it becomes necessary.
    if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
       && s_freeSessionSlots == 1)
    {
        // See if the gap is at maximum
        if(    (CONTEXT_SLOT)gr.contextCounter
            == gr.contextArray[s_oldestSavedSession])

            // Note: if this is being used on a TPM.combined, this return
            //       code should be transformed to an appropriate
            //       ISO/IEC 11889 (first edition) error code for this case.
            return TPM_RC_CONTEXT_GAP;
    }

    // Find an unoccupied entry in the contextArray
    for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
    {
        if(gr.contextArray[*handle] == 0)
        {
            // indicate that the session associated with this handle
            // references a loaded session
            gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex+1);
            return TPM_RC_SUCCESS;
        }
    }
    return TPM_RC_SESSION_HANDLES;
}
```

9.8.6.2 SessionCreate()

This function does the detailed work for starting an authorization session. This is done in a support routine rather than in the action code because the session management may differ in implementations. This implementation uses a fixed memory allocation to hold sessions and a fixed allocation to hold the *contextID* for the saved contexts.

Table 83

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	need to recycle sessions
TPM_RC_SESSION_HANDLE	active session space is full
TPM_RC_SESSION_MEMORY	loaded session space is full

```
TPM_RC
SessionCreate(
    TPM_SE            sessionType,    // IN: the session type
    TPMI_ALG_HASH     authHash,       // IN: the hash algorithm
    TPM2B_NONCE      *nonceCaller,    // IN: initial nonceCaller
    TPMT_SYM_DEF     *symmetric,      // IN: the symmetric algorithm
```

```
    TPMI_DH_ENTITY   bind,           // IN: the bind object
    TPM2B_DATA      *seed,           // IN: seed data
    TPM_HANDLE      *sessionHandle   // OUT: the session handle
    )
{
    TPM_RC                 result = TPM_RC_SUCCESS;
    CONTEXT_SLOT           slotIndex;
    SESSION               *session = NULL;

    pAssert(   sessionType == TPM_SE_HMAC
            || sessionType == TPM_SE_POLICY
            || sessionType == TPM_SE_TRIAL);

    // If there are no open spots in the session array, then no point in searching
    if(s_freeSessionSlots == 0)
        return TPM_RC_SESSION_MEMORY;

    // Find a space for loading a session
    for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
    {
        // Is this available?
        if(s_sessions[slotIndex].occupied == FALSE)
        {
            session = &s_sessions[slotIndex].session;
            break;
        }
    }
    // if no spot found, then this is an internal error
    pAssert (slotIndex < MAX_LOADED_SESSIONS);

    // Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
    // returned from ContextIdHandelAssign()
    result = ContextIdSessionCreate(sessionHandle, slotIndex);
    if(result != TPM_RC_SUCCESS)
        return result;

    //*** Only return from this point on is TPM_RC_SUCCESS

    // Can now indicate that the session array entry is occupied.
    s_freeSessionSlots--;
    s_sessions[slotIndex].occupied = TRUE;

    // Initialize the session data
    MemorySet(session, 0, sizeof(SESSION));

    // Initialize internal session data
    session->authHashAlg = authHash;
    // Initialize session type
    if(sessionType == TPM_SE_HMAC)
    {
        *sessionHandle += HMAC_SESSION_FIRST;

    }
    else
    {
        *sessionHandle += POLICY_SESSION_FIRST;

        // For TPM_SE_POLICY or TPM_SE_TRIAL
        session->attributes.isPolicy = SET;
        if(sessionType == TPM_SE_TRIAL)
            session->attributes.isTrialPolicy = SET;

        // Initialize policy session data
        SessionInitPolicyData(session);
    }
    // Create initial session nonce
```

```
    session->nonceTPM.t.size = nonceCaller->t.size;
    CryptGenerateRandom(session->nonceTPM.t.size, session->nonceTPM.t.buffer);

    // Set up session parameter encryption algorithm
    session->symmetric = *symmetric;

    // If there is a bind object or a session secret, then need to compute
    // a sessionKey.
    if(bind != TPM_RH_NULL || seed->t.size != 0)
    {
        // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
        //                      nonceCaller, bits)
        // The HMAC key for generating the sessionSecret can be the concatenation
        // of an authorization value and a seed value
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
        // label values.

        TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
        TPM2B_KEY            key;

        UINT16               hashSize;      // The size of the hash used by the
                                            // session crated by this command
        TPM2B_AUTH  entityAuth;             // The authValue of the entity
                                            // associated with HMAC session

        // Get hash size, which is also the length of sessionKey
        hashSize = CryptGetHashDigestSize(session->authHashAlg);

        // Get authValue of associated entity
        entityAuth.t.size = EntityGetAuthValue(bind, &entityAuth.t.buffer);

        // Concatenate authValue and seed
        pAssert(entityAuth.t.size + seed->t.size <= <K>sizeof(key.t.buffer));
        MemoryCopy2B(&key.b, &entityAuth.b, sizeof(key.t.buffer));
        MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));

        session->sessionKey.t.size = hashSize;

        // Compute the session key
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
        // label values.
        KDFa(session->authHashAlg, &key.b, "ATH", &session->nonceTPM.b,
             &nonceCaller->b, hashSize * 8, session->sessionKey.t.buffer, NULL);
    }

    // Copy the name of the entity that the HMAC session is bound to
    // Policy session is not bound to an entity
    if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
    {
        session->attributes.isBound = SET;
        SessionComputeBoundEntity(bind, &session->u1.boundEntity);
    }
    // If there is a bind object and it is subject to DA, then use of this session
    // is subject to DA regardless of how it is used.
    session->attributes.isDaBound =    (bind != TPM_RH_NULL)
                                    && (IsDAExempted(bind) == FALSE);

    // If the session is bound, then check to see if it is bound to lockoutAuth
    session->attributes.isLockoutBound =    (session->attributes.isDaBound  == SET)
                                         && (bind == TPM_RH_LOCKOUT);
    return TPM_RC_SUCCESS;

}
```

9.8.6.3 SessionContextSave()

This function is called when a session context is to be saved. The *contextID* of the saved session is returned. If no *contextID* can be assigned, then the routine returns TPM_RC_CONTEXT_GAP. If the function completes normally, the session slot will be freed.

This function requires that *handle* references a loaded session. Otherwise, it should not be called at the first place.

Table 84

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned.
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```
TPM_RC
SessionContextSave (
    TPM_HANDLE           handle,        // IN: session handle
    CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
    )
{
    UINT32                      contextIndex;
    CONTEXT_SLOT                slotIndex;

    pAssert(SessionIsLoaded(handle));

    // check to see if the gap is already maxed out
    // Need to have a saved session
    if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
          // if the oldest saved session has the same value as the low bits
          // of the contextCounter, then the GAP is maxed out.
       && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
        return TPM_RC_CONTEXT_GAP;

    // if the caller wants the context counter, set it
    if(contextID != NULL)
        *contextID = gr.contextCounter;

    pAssert((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS);

    contextIndex = handle & HR_HANDLE_MASK;

    // Extract the session slot number referenced by the contextArray
    // because we are going to overwrite this with the low order
    // contextID value.
    slotIndex = gr.contextArray[contextIndex] - 1;

    // Set the contextID for the contextArray
    gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;

    // Increment the counter
    gr.contextCounter++;

    // In the unlikely event that the 64-bit context counter rolls over...
    if(gr.contextCounter == 0)
    {
        // back it up
        gr.contextCounter--;
        // return an error
        return TPM_RC_TOO_MANY_CONTEXTS;
    }
    // if the low-order bits wrapped, need to advance the value to skip over
```



```
    // the values used to indicate that a session is loaded
    if(((CONTEXT_SLOT)gr.contextCounter) == 0)
        gr.contextCounter += MAX_LOADED_SESSIONS + 1;

    // If no other sessions are saved, this is now the oldest.
    if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
        s_oldestSavedSession = contextIndex;

    // Mark the session slot as unoccupied
    s_sessions[slotIndex].occupied = FALSE;

    // and indicate that there is an additional open slot
    s_freeSessionSlots++;

    return TPM_RC_SUCCESS;
}
```

9.8.6.4 SessionContextLoad()

This function is used to load a session from saved context. The session handle must be for a saved context.

If the gap is at a maximum, then the only session that can be loaded is the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned.

This function requires that *handle* references a valid saved session.

Table 85

Error Returns	Meaning
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_CONTEXT_GAP	the gap count is maximum and this is not the oldest saved context

```
TPM_RC
SessionContextLoad(
    SESSION         *session,       // IN: session structure from saved context
    TPM_HANDLE      *handle         // IN/OUT: session handle
    )
{
    UINT32                  contextIndex;
    CONTEXT_SLOT            slotIndex;

    pAssert(    HandleGetType(*handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);

    // Don't bother looking if no openings
    if(s_freeSessionSlots == 0)
        return TPM_RC_SESSION_MEMORY;

    // Find a free session slot to load the session
    for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
        if(s_sessions[slotIndex].occupied == FALSE) break;

    // if no spot found, then this is an internal error
    pAssert (slotIndex < MAX_LOADED_SESSIONS);

    contextIndex = *handle & HR_HANDLE_MASK;   // extract the index

    // If there is only one slot left, and the gap is at maximum, the only session
    // context that we can safely load is the oldest one.
    if(   s_oldestSavedSession < MAX_ACTIVE_SESSIONS
       && s_freeSessionSlots == 1
```

```
       && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
       && contextIndex != s_oldestSavedSession
      )
        return TPM_RC_CONTEXT_GAP;

    pAssert(contextIndex < MAX_ACTIVE_SESSIONS);

    // set the contextArray value to point to the session slot where
    // the context is loaded
    gr.contextArray[contextIndex] = slotIndex + 1;

    // if this was the oldest context, find the new oldest
    if(contextIndex == s_oldestSavedSession)
        ContextIdSetOldest();

    // Copy session data to session slot
    s_sessions[slotIndex].session = *session;

    // Set session slot as occupied
    s_sessions[slotIndex].occupied = TRUE;

    // Reduce the number of open spots
    s_freeSessionSlots--;

    return TPM_RC_SUCCESS;
}
```

9.8.6.5 SessionFlush()

This function is used to flush a session referenced by its handle. If the session associated with *handle* is loaded, the session array entry is marked as available.

This function requires that *handle* be a valid active session.

```
void
SessionFlush(
    TPM_HANDLE       handle         // IN: loaded or saved session handle
    )
{
    CONTEXT_SLOT         slotIndex;
    UINT32               contextIndex;   // Index into contextArray

    pAssert(   (    HandleGetType(handle) == TPM_HT_POLICY_SESSION
                 || HandleGetType(handle) == TPM_HT_HMAC_SESSION
               )
            && (SessionIsLoaded(handle)  || SessionIsSaved(handle))
           );

    // Flush context ID of this session
    // Convert handle to an index into the contextArray
    contextIndex = handle & HR_HANDLE_MASK;

    pAssert(contextIndex < <K>sizeof(gr.contextArray)/sizeof(gr.contextArray[0]));

    // Get the current contents of the array
    slotIndex = gr.contextArray[contextIndex];

    // Mark context array entry as available
    gr.contextArray[contextIndex] = 0;

    // Is this a saved session being flushed
    if(slotIndex > MAX_LOADED_SESSIONS)
    {
        // Flushing the oldest session?
```



```
        if(contextIndex == s_oldestSavedSession)
            // If so, find a new value for oldest.
            ContextIdSetOldest();
    }
    else
    {
        // Adjust slot index to point to session array index
        slotIndex -= 1;

        // Free session array index
        s_sessions[slotIndex].occupied = FALSE;
        s_freeSessionSlots++;
    }

    return;
}
```

9.8.6.6 SessionComputeBoundEntity()

This function computes the binding value for a session. The binding value for a reserved handle is the handle itself. For all the other entities, the *authValue* at the time of binding is included to prevent squatting. For those values, the Name and the *authValue* are concatenated into the bind buffer. If they will not both fit, the will be overlapped by XORing() bytes. If XOR is required, the bind value will be full.

```
void
SessionComputeBoundEntity(
    TPMI_DH_ENTITY   entityHandle,  // IN: handle of entity
    TPM2B_NAME      *bind           // OUT: binding value
    )
{
    TPM2B_AUTH          auth;
    INT16               overlap;

    // Get name
    bind->t.size = EntityGetName(entityHandle, &bind->t.name);

//    // The bound value of a reserved handle is the handle itself
//    if(bind->t.size == sizeof(TPM_HANDLE)) return;

    // For all the other entities, concatenate the auth value to the name.
    // Get a local copy of the auth value because some overlapping
    // may be necessary.
    auth.t.size = EntityGetAuthValue(entityHandle, &auth.t.buffer);
    pAssert(auth.t.size <= <K>sizeof(TPMU_HA));

    // Figure out if there will be any overlap
    overlap = bind->t.size + auth.t.size - sizeof(bind->t.name);

    // There is overlap if the combined sizes are greater than will fit
    if(overlap > 0)
    {
        // The overlap area is at the end of the Name
        BYTE    *result = &bind->t.name[bind->t.size - overlap];
        int     i;

        // XOR the auth value into the Name for the overlap area
        for(i = 0; i < overlap; i++)
            result[i] ^= auth.t.buffer[i];
    }
    else
    {
        // There is no overlap
        overlap = 0;
    }
```

```
    //copy the remainder of the authData to the end of the name
    MemoryCopy(&bind->t.name[bind->t.size], &auth.t.buffer[overlap],
               auth.t.size - overlap, sizeof(bind->t.name) - bind->t.size);

    // Increase the size of the bind data by the size of the auth - the overlap
    bind->t.size += auth.t.size-overlap;

    return;
}
```

9.8.6.7 SessionInitPolicyData()

This function initializes the portions of the session policy data that are not set by the allocation of a session.

```
void
SessionInitPolicyData(
    SESSION         *session        // IN: session handle
    )
{
    // Initialize start time
    session->startTime = go.clock;

    // Initialize policyDigest.  policyDigest is initialized with a string of 0 of
    // session algorithm digest size. Since the policy already contains all zeros
    // it is only necessary to set the size
    session->u2.policyDigest.t.size = CryptGetHashDigestSize(session->authHashAlg);
    return;
}
```

9.8.6.8 SessionResetPolicyData()

This function is used to reset the policy data without changing the nonce or the start time of the session.

```
void
SessionResetPolicyData(
    SESSION         *session        // IN: the session to reset
    )
{
    session->commandCode = 0;      // No command

    // No locality selected
    MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));

    // The cpHash size to zero
    session->u1.cpHash.b.size = 0;

    // No timeout
    session->timeOut = 0;

    // Reset the pcrCounter
    session->pcrCounter = 0;

    // Reset the policy hash
    MemorySet(&session->u2.policyDigest.t.buffer, 0,
              session->u2.policyDigest.t.size);

    // Reset the session attributes
    MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));

    // set the policy attribute
    session->attributes.isPolicy = SET;
}
```


9.8.6.9 SessionCapGetLoaded()

This function returns a list of handles of loaded session, started from input *handle*

Handle must be in valid loaded session handle range, but does not have to point to a loaded session.

Table 86

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
SessionCapGetLoaded(
    TPMI_SH_POLICY    handle,         // IN: start handle
    UINT32            count,          // IN: count of returned handles
    TPML_HANDLE      *handleList      // OUT: list of handle
    )
{
    TPMI_YES_NO      more = NO;
    UINT32           i;

    pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate session context ID slots to get loaded session handles
    for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If session is active
        if(gr.contextArray[i] != 0)
        {
            // If session is loaded
            if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
            {
                if(handleList->count < count)
                {
                    SESSION         *session;

                    // If we have not filled up the return list, add this
                    // session handle to it
                    // assume that this is going to be an HMAC session
                    handle = i + HMAC_SESSION_FIRST;
                    session = SessionGet(handle);
                    if(session->attributes.isPolicy)
                        handle = i + POLICY_SESSION_FIRST;
                    handleList->handle[handleList->count] = handle;
                    handleList->count++;
                }
                else
                {
                    // If the return list is full but we still have loaded object
                    // available, report this and stop iterating
                    more = YES;
                    break;
                }
            }
        }
    }
```

```
    return more;

}
```

9.8.6.10 SessionCapGetSaved()

This function returns a list of handles for saved session, starting at *handle*.

Handle must be in a valid handle range, but does not have to point to a saved session.

Table 87

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
SessionCapGetSaved(
    TPMI_SH_HMAC      handle,         // IN: start handle
    UINT32            count,          // IN: count of returned handles
    TPML_HANDLE      *handleList      // OUT: list of handle
    )
{
    TPMI_YES_NO      more = NO;
    UINT32           i;

    pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate session context ID slots to get loaded session handles
    for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If session is active
        if(gr.contextArray[i] != 0)
        {
            // If session is saved
            if (gr.contextArray[i] > MAX_LOADED_SESSIONS)
            {
                if(handleList->count < count)
                {
                    // If we have not filled up the return list, add this
                    // session handle to it
                    handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
                    handleList->count++;
                }
                else
                {
                    // If the return list is full but we still have loaded object
                    // available, report this and stop iterating
                    more = YES;
                    break;
                }
            }
        }
    }

```



```
    return more;

}
```

9.8.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

```
UINT32
SessionCapGetLoadedNumber(
    void
    )
{
    return MAX_LOADED_SESSIONS - s_freeSessionSlots;
}
```

9.8.6.12 SessionCapGetLoadedAvail()

This function returns the number of additional authorization sessions, of any type, that could be loaded into TPM RAM.

NOTE In other implementations, this number might just be an estimate. The only constraint for the estimate is, if it is one or more, then at least one session needs to be loadable.

```
UINT32
SessionCapGetLoadedAvail(
    void
    )
{
    return s_freeSessionSlots;
}
```

9.8.6.13 SessionCapGetActiveNumber()

This function returns the number of active authorization sessions currently being tracked by the TPM.

```
UINT32
SessionCapGetActiveNumber(
    void
    )
{
    UINT32              i;
    UINT32              num = 0;

    // Iterate the context array to find the number of non-zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] != 0) num++;
    }

    return num;
}
```

9.8.6.14 SessionCapGetActiveAvail()

This function returns the number of additional authorization sessions, of any type, that could be created. This not the number of slots for sessions, but the number of additional sessions that the TPM is capable of tracking.

```
UINT32
SessionCapGetActiveAvail(
    void
    )
{
    UINT32              i;
    UINT32              num = 0;

    // Iterate the context array to find the number of zero slots
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        if(gr.contextArray[i] == 0) num++;
    }

    return num;
}
```

9.9 Time.c

9.9.1 Introduction

This file contains the functions relating to the TPM's time functions including the interface to the implementation-specific time functions.

9.9.2 Includes

```
#include "InternalRoutines.h"
#include "Platform.h"
```

9.9.3 Functions

9.9.3.1 TimePowerOn()

This function initialize time info at _TPM_Init().

```
void
TimePowerOn(
    void
    )
{
    TPM_SU              orderlyShutDown;

    // Read orderly data info from NV memory
    NvReadReserved(NV_ORDERLY_DATA, &go);

    // Read orderly shut down state flag
    NvReadReserved(NV_ORDERLY, &orderlyShutDown);

    // If the previous cycle is orderly shut down, the value of the safe bit
    // the same as previously saved.  Otherwise, it is not safe.
    if(orderlyShutDown == SHUTDOWN_NONE)
        go.clockSafe= NO;
    else
        go.clockSafe = YES;

    // Set the initial state of the DRBG
    CryptDrbgGetPutState(PUT_STATE);

    // Clear time since TPM power on
    g_time = 0;
```

```

    return;
}
```

9.9.3.2 TimeStartup()

This function updates the *resetCount* and *restartCount* components of TPMS_CLOCK_INFO structure at TPM2_Startup().

```
void
TimeStartup(
    STARTUP_TYPE     type           // IN: start up type
    )
{
    if(type == SU_RESUME)
    {
        // Resume sequence
        gr.restartCount++;
    }
    else
    {
        if(type == SU_RESTART)
        {
            // Hibernate sequence
            gr.clearCount++;
            gr.restartCount++;
        }
        else
        {
            // Reset sequence
            // Increase resetCount
            gp.resetCount++;

            // Write resetCount to NV
            NvWriteReserved(NV_RESET_COUNT, &gp.resetCount);
            gp.totalResetCount++;

            // We do not expect the total reset counter overflow during the life
            // time of TPM.  if it ever happens, TPM will be put to failure mode
            // and there is no way to recover it.
            // The reason that there is no recovery is that we don't increment
            // the NV totalResetCount when incrementing would make it 0. When the
            // TPM starts up again, the old value of totalResetCount will be read
            // and we will get right back to here with the increment failing.
            if(gp.totalResetCount == 0)
                FAIL(FATAL_ERROR_INTERNAL);


            // Write total reset counter to NV
            NvWriteReserved(NV_TOTAL_RESET_COUNT, &gp.totalResetCount);

            // Reset restartCount
            gr.restartCount = 0;
        }
    }

    return;
}
```


9.9.3.3 TimeUpdateToCurrent()

This function updates the *Time* and *Clock* in the global TPMS_TIME_INFO structure.

In this implementation, *Time* and *Clock* are updated at the beginning of each command and the values are unchanged for the duration of the command.

Because *Clock* updates may require a write to NV memory, *Time* and *Clock* are not allowed to advance if NV is not available. When clock is not advancing, any function that uses *Clock* will fail and return TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE.

This implementations does not do rate limiting. If the implementation does do rate limiting, then the *Clock* update should not be inhibited even when doing rather limiting.

```
void
TimeUpdateToCurrent(
    void
    )
{
    UINT64          oldClock;
    UINT64          elapsed;
#define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)

    // Can't update time during the dark interval or when rate limiting.
    if(NvIsAvailable() != TPM_RC_SUCCESS)
        return;

    // Save the old clock value
    oldClock = go.clock;

    // Update the time info to current
    elapsed = _plat__ClockTimeElapsed();
    go.clock += elapsed;
    g_time += elapsed;

    // Check to see if the update has caused a need for an nvClock update
    // CLOCK_UPDATE_MASK is measured by second, while the value in go.clock is
    // recorded by millisecond.  Align the clock value to second before the bit
    // operations
    if( ((go.clock/1000) | CLOCK_UPDATE_MASK)
            > ((oldClock/1000) | CLOCK_UPDATE_MASK))
    {
        // Going to update the time state so the safe flag
        // should be set
        go.clockSafe = YES;

        // Get the DRBG state before updating orderly data
        CryptDrbgGetPutState(GET_STATE);

        NvWriteReserved(NV_ORDERLY_DATA, &go);
    }

    // Call self healing logic for dictionary attack parameters
    DASelfHeal();

    return;
}
```


9.9.3.4 TimeSetAdjustRate()

This function is used to perform rate adjustment on *Time* and *Clock*.

```
void
TimeSetAdjustRate(
    TPM_CLOCK_ADJUST     adjust         // IN: adjust constant
    )
{
    switch(adjust)
```



```
    {
        case TPM_CLOCK_COARSE_SLOWER:
            _plat__ClockAdjustRate(CLOCK_ADJUST_COARSE);
            break;
        case TPM_CLOCK_COARSE_FASTER:
            _plat__ClockAdjustRate(-CLOCK_ADJUST_COARSE);
            break;
        case TPM_CLOCK_MEDIUM_SLOWER:
            _plat__ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
            break;
        case TPM_CLOCK_MEDIUM_FASTER:
            _plat__ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
            break;
        case TPM_CLOCK_FINE_SLOWER:
            _plat__ClockAdjustRate(CLOCK_ADJUST_FINE);
            break;
        case TPM_CLOCK_FINE_FASTER:
            _plat__ClockAdjustRate(-CLOCK_ADJUST_FINE);
            break;
        case TPM_CLOCK_NO_CHANGE:
            break;
        default:
            pAssert(FALSE);
            break;
    }

    return;
}
```

9.9.3.5 TimeGetRange()

This function is used to access TPMS_TIME_INFO. The TPMS_TIME_INFO structure is treaded as an array of bytes, and a byte offset and length determine what bytes are returned.

Table 88

Error Returns	Meaning
TPM_RC_RANGE	invalid data range

```
TPM_RC
TimeGetRange(
    UINT16              offset,         // IN: offset in TPMS_TIME_INFO
    UINT16              size,           // IN: size of data
    TIME_INFO          *dataBuffer      // OUT: result buffer
    )
{
    TPMS_TIME_INFO      timeInfo;
    UINT16              infoSize;
    BYTE                infoData[sizeof(TPMS_TIME_INFO)];
    BYTE                *buffer;

    // Fill TPMS_TIME_INFO structure
    timeInfo.time = g_time;
    TimeFillInfo(&timeInfo.clockInfo);

    // Marshal TPMS_TIME_INFO to canonical form
    buffer = infoData;
    infoSize = TPMS_TIME_INFO_Marshal(&timeInfo, &buffer, NULL);

    // Check if the input range is valid
    if(offset + size > infoSize) return TPM_RC_RANGE;

```

```
    // Copy info data to output buffer
    MemoryCopy(dataBuffer, infoData + offset, size, sizeof(TIME_INFO));

    return TPM_RC_SUCCESS;
}
```

9.9.3.6 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```
void
TimeFillInfo(
    TPMS_CLOCK_INFO     *clockInfo
    )
{
    clockInfo->clock = go.clock;
    clockInfo->resetCount = gp.resetCount;
    clockInfo->restartCount = gr.restartCount;

    // If NV is not available, clock stopped advancing and the value reported is
    // not "safe".
    if(NvIsAvailable() == TPM_RC_SUCCESS)
        clockInfo->safe = go.clockSafe;
    else
        clockInfo->safe = NO;

    return;
}
```


10 Support

10.1 AlgorithmCap.c

10.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2_GetCapability() to return the algorithm properties.

10.1.2 Includes and Defines

```
#include "InternalRoutines.h"
typedef struct
{
    TPM_ALG_ID          algID;
    TPMA_ALGORITHM      attributes;
} ALGORITHM;
static const ALGORITHM    s_algorithms[] =
{
#ifdef TPM_ALG_RSA
    {TPM_ALG_RSA,           {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA1
    {TPM_ALG_SHA1,          {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_HMAC
    {TPM_ALG_HMAC,          {0, 0, 1, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_AES
    {TPM_ALG_AES,           {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_MGF1
    {TPM_ALG_MGF1,          {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif

    {TPM_ALG_KEYEDHASH,     {0, 0, 1, 1, 0, 1, 1, 0, 0}},

#ifdef TPM_ALG_XOR
    {TPM_ALG_XOR,           {0, 1, 1, 0, 0, 0, 0, 0, 0}},
#endif

#ifdef TPM_ALG_SHA256
    {TPM_ALG_SHA256,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA384
    {TPM_ALG_SHA384,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SHA512
    {TPM_ALG_SHA512,        {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_WHIRLPOOL512
    {TPM_ALG_WHIRLPOOL512,  {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM3_256
    {TPM_ALG_SM3_256,       {0, 0, 1, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_SM4
    {TPM_ALG_SM4,          {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_RSASSA
    {TPM_ALG_RSASSA,        {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
```

```
#ifdef TPM_ALG_RSAES
    {TPM_ALG_RSAES,          {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_RSAPSS
    {TPM_ALG_RSAPSS,         {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_OAEP
    {TPM_ALG_OAEP,           {1, 0, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_ECDSA
    {TPM_ALG_ECDSA,          {1, 0, 0, 0, 0, 1, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECDH
    {TPM_ALG_ECDH,           {1, 0, 0, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECDAA
    {TPM_ALG_ECDAA,          {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_ECSCHNORR
    {TPM_ALG_ECSCHNORR,      {1, 0, 0, 0, 0, 1, 0, 0, 0}},
#endif
#ifdef TPM_ALG_KDF1_SP800_56a
    {TPM_ALG_KDF1_SP800_56a,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_KDF2
    {TPM_ALG_KDF2,           {0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_KDF1_SP800_108
    {TPM_ALG_KDF1_SP800_108,{0, 0, 1, 0, 0, 0, 0, 1, 0}},
#endif
#ifdef TPM_ALG_ECC
    {TPM_ALG_ECC,            {1, 0, 0, 1, 0, 0, 0, 0, 0}},
#endif

    {TPM_ALG_SYMCIPHER,      {0, 0, 0, 1, 0, 0, 0, 0, 0}},

#ifdef TPM_ALG_CAMELLIA
    {TPM_ALG_CAMELLIA,       {0, 1, 0, 0, 0, 0, 0, 0, 0}},
#endif
#ifdef TPM_ALG_CTR
    {TPM_ALG_CTR,            {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_OFB
    {TPM_ALG_OFB,            {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_CBC
    {TPM_ALG_CBC,            {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_CFB
    {TPM_ALG_CFB,            {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
#ifdef TPM_ALG_ECB
    {TPM_ALG_ECB,            {0, 1, 0, 0, 0, 0, 1, 0, 0}},
#endif
};
```

10.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

Table 89

Return Value	Meaning
YES	more algorithms to report
NO	no more algorithms to report

```
TPMI_YES_NO
AlgorithmCapGetImplemented(
    TPM_ALG_ID                      algID,      // IN: the starting algorithm ID
    UINT32                          count,      // IN: count of returned algorithms
    TPML_ALG_PROPERTY              *algList     // OUT: algorithm list
)
{
    TPMI_YES_NO     more = NO;
    UINT32          i;
    UINT32          algNum;

    // initialize output algorithm list
    algList->count = 0;

    // The maximum count of algorithms we may return is MAX_CAP_ALGS.
    if(count > MAX_CAP_ALGS)
        count = MAX_CAP_ALGS;

    // Compute how many algorithms are defined in s_algorithms array.
    algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);

    // Scan the implemented algorithm list to see if there is a match to 'algID'.
    for(i = 0; i < algNum; i++)
    {
        // If algID is less than the starting algorithm ID, skip it
        if(s_algorithms[i].algID < algID)
            continue;
        if(algList->count < count)
        {
            // If we have not filled up the return list, add more algorithms
            // to it
            algList->algProperties[algList->count].alg = s_algorithms[i].algID;
            algList->algProperties[algList->count].algProperties =
                s_algorithms[i].attributes;
            algList->count++;
        }
        else
        {
            // If the return list is full but we still have algorithms
            // available, report this and stop scanning.
            more = YES;
            break;
        }

    }

    return more;

}
LIB_EXPORT
void
AlgorithmGetImplementedVector(
    ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
    )
{
    int                         index;

    // Nothing implemented until we say it is
```

```
    MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));

    for(index =  (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
        index >= 0;
        index--)
            SET_BIT(s_algorithms[index].algID, *implemented);
    return;
}
```

10.2 Bits.c

10.2.1 Introduction

This file contains bit manipulation routines. They operate on bit arrays.

The 0th bit in the array is the right-most bit in the 0th octet in the array.

NOTE If pAssert() is defined, the functions will assert if the indicated bit number is outside of the range of *bArray*. How the assert is handled is implementation dependent.

10.2.2 Includes

```
#include "InternalRoutines.h"
```

10.2.3 Functions

10.2.3.1 BitIsSet()

This function is used to check the setting of a bit in an array of bits.

Table 90

Return Value	Meaning
TRUE	bit is set
FALSE	bit is not set

```
BOOL
BitIsSet(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'
    BYTE            *bArray,        // IN: array containing the bits
    unsigned int     arraySize      // IN: size in bytes of 'bArray'
    )
{
    pAssert(arraySize > (bitNum >> 3));
    return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
}
```

10.2.3.2 BitSet()

This function will set the indicated bit in *bArray*.

```
void
BitSet(
    unsigned int     bitNum,        // IN: number of the bit in 'bArray'
    BYTE            *bArray,        // IN: array containing the bits
    unsigned int     arraySize      // IN: size in bytes of 'bArray'
```



```
    )
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
```

10.2.3.3 BitClear()

This function will clear the indicated bit in *bArray*.

```
void
BitClear(
    unsigned int      bitNum,        // IN: number of the bit in 'bArray'.
    BYTE             *bArray,        // IN: array containing the bits
    unsigned int      arraySize      // IN: size in bytes of 'bArray'
    )
{
    pAssert(arraySize > bitNum/8);
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
```

10.3 CommandAttributeData.c

This is the command code attribute array for GetCapability(). Both this array and *s_commandAttributes* provides command code attributes, but tuned for different purpose.

```
static const TPMA_CC    s_ccAttr [] = {
        {0x011f, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_UndefineSpaceSpecial
        {0x0120, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_EvictControl
        {0x0121, 0, 1, 1, 0, 1, 0, 0, 0},      // TPM_CC_HierarchyControl
        {0x0122, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_UndefineSpace
        {0x0123, 0, 0, 0, 0, 0, 0, 0, 0},      // No command
        {0x0124, 0, 1, 1, 0, 1, 0, 0, 0},      // TPM_CC_ChangeEPS
        {0x0125, 0, 1, 1, 0, 1, 0, 0, 0},      // TPM_CC_ChangePPS
        {0x0126, 0, 1, 1, 0, 1, 0, 0, 0},      // TPM_CC_Clear
        {0x0127, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_ClearControl
        {0x0128, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_ClockSet
        {0x0129, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_HierarchyChangeAuth
        {0x012a, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_NV_DefineSpace
        {0x012b, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_Allocate
        {0x012c, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_SetAuthPolicy
        {0x012d, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PP_Commands
        {0x012e, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_SetPrimaryPolicy
        {0x012f, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_FieldUpgradeStart
        {0x0130, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ClockRateAdjust
        {0x0131, 0, 0, 0, 0, 1, 1, 0, 0},      // TPM_CC_CreatePrimary
        {0x0132, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_NV_GlobalWriteLock
        {0x0133, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_GetCommandAuditDigest
        {0x0134, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_Increment
        {0x0135, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_SetBits
        {0x0136, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_Extend
        {0x0137, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_Write
        {0x0138, 0, 1, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_WriteLock
        {0x0139, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_DictionaryAttackLockReset
        {0x013a, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_DictionaryAttackParameters
        {0x013b, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_NV_ChangeAuth
        {0x013c, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_Event
        {0x013d, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_Reset
        {0x013e, 0, 0, 0, 1, 1, 0, 0, 0},      // TPM_CC_SequenceComplete
        {0x013f, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_SetAlgorithmSet
        {0x0140, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_SetCommandCodeAuditStatus
        {0x0141, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_FieldUpgradeData
        {0x0142, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_IncrementalSelfTest
```

```
        {0x0143, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_SelfTest
        {0x0144, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_Startup
        {0x0145, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_Shutdown
        {0x0146, 0, 1, 0, 0, 0, 0, 0, 0},      // TPM_CC_StirRandom
        {0x0147, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_ActivateCredential
        {0x0148, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_Certify
        {0x0149, 0, 0, 0, 0, 3, 0, 0, 0},      // TPM_CC_PolicyNV
        {0x014a, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_CertifyCreation
        {0x014b, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_Duplicate
        {0x014c, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_GetTime
        {0x014d, 0, 0, 0, 0, 3, 0, 0, 0},      // TPM_CC_GetSessionAuditDigest
        {0x014e, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_Read
        {0x014f, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_NV_ReadLock
        {0x0150, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_ObjectChangeAuth
        {0x0151, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_PolicySecret
        {0x0152, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_Rewrap
        {0x0153, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Create
        {0x0154, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ECDH_ZGen
        {0x0155, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_HMAC
        {0x0156, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Import
        {0x0157, 0, 0, 0, 0, 1, 1, 0, 0},      // TPM_CC_Load
        {0x0158, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Quote
        {0x0159, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_RSA_Decrypt
        {0x015a, 0, 0, 0, 0, 0, 0, 0, 0},      // No command
        {0x015b, 0, 0, 0, 0, 1, 1, 0, 0},      // TPM_CC_HMAC_Start
        {0x015c, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_SequenceUpdate
        {0x015d, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Sign
        {0x015e, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Unseal
        {0x015f, 0, 0, 0, 0, 0, 0, 0, 0},      // No command
        {0x0160, 0, 0, 0, 0, 2, 0, 0, 0},      // TPM_CC_PolicySigned
        {0x0161, 0, 0, 0, 0, 0, 1, 0, 0},      // TPM_CC_ContextLoad
        {0x0162, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ContextSave
        {0x0163, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ECDH_KeyGen
        {0x0164, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_EncryptDecrypt
        {0x0165, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_FlushContext
        {0x0166, 0, 0, 0, 0, 0, 0, 0, 0},      // No command
        {0x0167, 0, 0, 0, 0, 0, 1, 0, 0},      // TPM_CC_LoadExternal
        {0x0168, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_MakeCredential
        {0x0169, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_NV_ReadPublic
        {0x016a, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyAuthorize
        {0x016b, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyAuthValue
        {0x016c, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyCommandCode
        {0x016d, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyCounterTimer
        {0x016e, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyCpHash
        {0x016f, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyLocality
        {0x0170, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyNameHash
        {0x0171, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyOR
        {0x0172, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyTicket
        {0x0173, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ReadPublic
        {0x0174, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_RSA_Encrypt
        {0x0175, 0, 0, 0, 0, 0, 0, 0, 0},      // No command
        {0x0176, 0, 0, 0, 0, 2, 1, 0, 0},      // TPM_CC_StartAuthSession
        {0x0177, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_VerifySignature
        {0x0178, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_ECC_Parameters
        {0x0179, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_FirmwareRead
        {0x017a, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_GetCapability
        {0x017b, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_GetRandom
        {0x017c, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_GetTestResult
        {0x017d, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_Hash
        {0x017e, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_PCR_Read
        {0x017f, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyPCR
        {0x0180, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyRestart
        {0x0181, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_ReadClock
        {0x0182, 0, 1, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_Extend
        {0x0183, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PCR_SetAuthValue
        {0x0184, 0, 0, 0, 0, 3, 0, 0, 0},      // TPM_CC_NV_Certify
```

```
        {0x0185, 0, 1, 0, 1, 2, 0, 0, 0},      // TPM_CC_EventSequenceComplete
        {0x0186, 0, 0, 0, 0, 0, 1, 0, 0},      // TPM_CC_HashSequenceStart
        {0x0187, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyPhysicalPresence
        {0x0188, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyDuplicationSelect
        {0x0189, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyGetDigest
        {0x018a, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_TestParms
        {0x018b, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_Commit
        {0x018c, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_PolicyPassword
        {0x018d, 0, 0, 0, 0, 1, 0, 0, 0},      // TPM_CC_ZGen_2Phase
        {0x018e, 0, 0, 0, 0, 0, 0, 0, 0},      // TPM_CC_EC_Ephemeral
        {0x018f, 0, 0, 0, 0, 1, 0, 0, 0}       // TPM_CC_PolicyNvWritten
};
typedef  UINT16             _ATTR_;
#define  NOT_IMPLEMENTED    (_ATTR_)(0)
#define  ENCRYPT_2         (_ATTR_)(1 <<  0)
#define  ENCRYPT_4         (_ATTR_)(1 <<  1)
#define  DECRYPT_2         (_ATTR_)(1 <<  2)
#define  DECRYPT_4         (_ATTR_)(1 <<  3)
#define  HANDLE_1_USER     (_ATTR_)(1 <<  4)
#define  HANDLE_1_ADMIN    (_ATTR_)(1 <<  5)
#define  HANDLE_1_DUP      (_ATTR_)(1 <<  6)
#define  HANDLE_2_USER     (_ATTR_)(1 <<  7)
#define  PP_COMMAND        (_ATTR_)(1 <<  8)
#define  IS_IMPLEMENTED    (_ATTR_)(1 <<  9)
#define  NO_SESSIONS       (_ATTR_)(1 << 10)
#define  NV_COMMAND        (_ATTR_)(1 << 11)
#define  PP_REQUIRED       (_ATTR_)(1 << 12)
#define  R_HANDLE          (_ATTR_)(1 << 13)
```

This is the command code attribute structure.

```
typedef UINT16 COMMAND_ATTRIBUTES;
static const COMMAND_ATTRIBUTES    s_commandAttributes [] = {
    (_ATTR_)(CC_NV_UndefineSpaceSpecial        *
(IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),                    // 0x011f
    (_ATTR_)(CC_EvictControl                   *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0120
    (_ATTR_)(CC_HierarchyControl               *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0121
    (_ATTR_)(CC_NV_UndefineSpace               *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0122
    (_ATTR_)                                     (NOT_IMPLEMENTED),
// 0x0123 - Not assigned
    (_ATTR_)(CC_ChangeEPS                      *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0124
    (_ATTR_)(CC_ChangePPS                      *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0125
    (_ATTR_)(CC_Clear                          *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0126
    (_ATTR_)(CC_ClearControl                   *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0127
    (_ATTR_)(CC_ClockSet                       *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x0128
    (_ATTR_)(CC_HierarchyChangeAuth            *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                         // 0x0129
    (_ATTR_)(CC_NV_DefineSpace                 *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                         // 0x012a
    (_ATTR_)(CC_PCR_Allocate                   *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                                   // 0x012b
    (_ATTR_)(CC_PCR_SetAuthPolicy              *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                         // 0x012c
    (_ATTR_)(CC_PP_Commands                    *
(IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),                                  // 0x012d
    (_ATTR_)(CC_SetPrimaryPolicy               *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),                         // 0x012e
```

```
    (_ATTR_)(CC_FieldUpgradeStart          * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),            // 0x012f
    (_ATTR_)(CC_ClockRateAdjust            * (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                       // 0x0130
    (_ATTR_)(CC_CreatePrimary              * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)), // 0x0131
    (_ATTR_)(CC_NV_GlobalWriteLock         * (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                       // 0x0132
    (_ATTR_)(CC_GetCommandAuditDigest      * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)), // 0x0133
    (_ATTR_)(CC_NV_Increment               * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x0134
    (_ATTR_)(CC_NV_SetBits                 * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x0135
    (_ATTR_)(CC_NV_Extend                  * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                        // 0x0136
    (_ATTR_)(CC_NV_Write                   * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                        // 0x0137
    (_ATTR_)(CC_NV_WriteLock               * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x0138
    (_ATTR_)(CC_DictionaryAttackLockReset  * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x0139
    (_ATTR_)(CC_DictionaryAttackParameters * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x013a
    (_ATTR_)(CC_NV_ChangeAuth              * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),                       // 0x013b
    (_ATTR_)(CC_PCR_Event                  * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                        // 0x013c
    (_ATTR_)(CC_PCR_Reset                  * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x013d
    (_ATTR_)(CC_SequenceComplete           * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),              // 0x013e
    (_ATTR_)(CC_SetAlgorithmSet            * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x013f
    (_ATTR_)(CC_SetCommandCodeAuditStatus  * (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),                       // 0x0140
    (_ATTR_)(CC_FieldUpgradeData           * (IS_IMPLEMENTED+DECRYPT_2)),              // 0x0141
    (_ATTR_)(CC_IncrementalSelfTest        * (IS_IMPLEMENTED)),                        // 0x0142
    (_ATTR_)(CC_SelfTest                   * (IS_IMPLEMENTED)),                        // 0x0143
    (_ATTR_)(CC_Startup                    * (IS_IMPLEMENTED+NO_SESSIONS)),            // 0x0144
    (_ATTR_)(CC_Shutdown                   * (IS_IMPLEMENTED)),                        // 0x0145
    (_ATTR_)(CC_StirRandom                 * (IS_IMPLEMENTED+DECRYPT_2)),              // 0x0146
    (_ATTR_)(CC_ActivateCredential         * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)), // 0x0147
    (_ATTR_)(CC_Certify                    * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)), // 0x0148
    (_ATTR_)(CC_PolicyNV                   * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                        // 0x0149
    (_ATTR_)(CC_CertifyCreation            * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),              // 0x014a
    (_ATTR_)(CC_Duplicate                  * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),               // 0x014b
    (_ATTR_)(CC_GetTime                    * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)), // 0x014c
    (_ATTR_)(CC_GetSessionAuditDigest      * (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)), // 0x014d
    (_ATTR_)(CC_NV_Read                    * (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                        // 0x014e
    (_ATTR_)(CC_NV_ReadLock                * (IS_IMPLEMENTED+HANDLE_1_USER)),          // 0x014f
```

```
    (_ATTR_)(CC_ObjectChangeAuth          *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),                          // 0x0150
    (_ATTR_)(CC_PolicySecret              *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0151
    (_ATTR_)(CC_Rewrap                    *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0152
    (_ATTR_)(CC_Create                    *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0153
    (_ATTR_)(CC_ECDH_ZGen                 *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0154
    (_ATTR_)(CC_HMAC                      *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0155
    (_ATTR_)(CC_Import                    *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0156
    (_ATTR_)(CC_Load                      *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),                  // 0x0157
    (_ATTR_)(CC_Quote                     *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0158
    (_ATTR_)(CC_RSA_Decrypt               *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                           // 0x0159
    (_ATTR_)                                (NOT_IMPLEMENTED),
// 0x015a - Not assigned
    (_ATTR_)(CC_HMAC_Start                *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),                            // 0x015b
    (_ATTR_)(CC_SequenceUpdate            *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                                     // 0x015c
    (_ATTR_)(CC_Sign                      *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                                     // 0x015d
    (_ATTR_)(CC_Unseal                    *
(IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                                     // 0x015e
    (_ATTR_)                                (NOT_IMPLEMENTED),
// 0x015f - Not assigned
    (_ATTR_)(CC_PolicySigned              * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
// 0x0160
    (_ATTR_)(CC_ContextLoad               * (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
// 0x0161
    (_ATTR_)(CC_ContextSave               * (IS_IMPLEMENTED+NO_SESSIONS)),
// 0x0162
    (_ATTR_)(CC_ECDH_KeyGen               * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x0163
    (_ATTR_)(CC_EncryptDecrypt            *
(IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),                                     // 0x0164
    (_ATTR_)(CC_FlushContext              * (IS_IMPLEMENTED+NO_SESSIONS)),
// 0x0165
    (_ATTR_)                                (NOT_IMPLEMENTED),
// 0x0166 - Not assigned
    (_ATTR_)(CC_LoadExternal              *
(IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                                // 0x0167
    (_ATTR_)(CC_MakeCredential            * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
// 0x0168
    (_ATTR_)(CC_NV_ReadPublic             * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x0169
    (_ATTR_)(CC_PolicyAuthorize           * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x016a
    (_ATTR_)(CC_PolicyAuthValue           * (IS_IMPLEMENTED)),
// 0x016b
    (_ATTR_)(CC_PolicyCommandCode         * (IS_IMPLEMENTED)),
// 0x016c
    (_ATTR_)(CC_PolicyCounterTimer        * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x016d
    (_ATTR_)(CC_PolicyCpHash              * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x016e
    (_ATTR_)(CC_PolicyLocality            * (IS_IMPLEMENTED)),
// 0x016f
    (_ATTR_)(CC_PolicyNameHash            * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x0170
```

```
    (_ATTR_)(CC_PolicyOR                    * (IS_IMPLEMENTED)),
// 0x0171
    (_ATTR_)(CC_PolicyTicket                * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x0172
    (_ATTR_)(CC_ReadPublic                  * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x0173
    (_ATTR_)(CC_RSA_Encrypt                 * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
// 0x0174
    (_ATTR_)                                  (NOT_IMPLEMENTED),
// 0x0175 - Not assigned
    (_ATTR_)(CC_StartAuthSession            *
(IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),                       // 0x0176
    (_ATTR_)(CC_VerifySignature             * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x0177
    (_ATTR_)(CC_ECC_Parameters              * (IS_IMPLEMENTED)),
// 0x0178
    (_ATTR_)(CC_FirmwareRead                * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x0179
    (_ATTR_)(CC_GetCapability               * (IS_IMPLEMENTED)),
// 0x017a
    (_ATTR_)(CC_GetRandom                   * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x017b
    (_ATTR_)(CC_GetTestResult               * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x017c
    (_ATTR_)(CC_Hash                        * (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
// 0x017d
    (_ATTR_)(CC_PCR_Read                    * (IS_IMPLEMENTED)),
// 0x017e
    (_ATTR_)(CC_PolicyPCR                   * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x017f
    (_ATTR_)(CC_PolicyRestart               * (IS_IMPLEMENTED)),
// 0x0180
    (_ATTR_)(CC_ReadClock                   * (IS_IMPLEMENTED+NO_SESSIONS)),
// 0x0181
    (_ATTR_)(CC_PCR_Extend                  * (IS_IMPLEMENTED+HANDLE_1_USER)),
// 0x0182
    (_ATTR_)(CC_PCR_SetAuthValue            *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),                            // 0x0183
    (_ATTR_)(CC_NV_Certify                  *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),    // 0x0184
    (_ATTR_)(CC_EventSequenceComplete       *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),              // 0x0185
    (_ATTR_)(CC_HashSequenceStart           * (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
// 0x0186
    (_ATTR_)(CC_PolicyPhysicalPresence      * (IS_IMPLEMENTED)),
// 0x0187
    (_ATTR_)(CC_PolicyDuplicationSelect     * (IS_IMPLEMENTED+DECRYPT_2)),
// 0x0188
    (_ATTR_)(CC_PolicyGetDigest             * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x0189
    (_ATTR_)(CC_TestParms                   * (IS_IMPLEMENTED)),
// 0x018a
    (_ATTR_)(CC_Commit                      *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                  // 0x018b
    (_ATTR_)(CC_PolicyPassword              * (IS_IMPLEMENTED)),
// 0x018c
    (_ATTR_)(CC_ZGen_2Phase                 *
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),                  // 0x018d
    (_ATTR_)(CC_EC_Ephemeral                * (IS_IMPLEMENTED+ENCRYPT_2)),
// 0x018e
    (_ATTR_)(CC_PolicyNvWritten             * (IS_IMPLEMENTED))
// 0x018f
};
```


10.4 CommandCodeAttributes.c

10.4.1 Introduction

This file contains the functions for testing various command properties.

10.4.2 Includes and Defines

```
#include     "Tpm.h"
#include     "InternalRoutines.h"
typedef UINT16           ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in ISO/IEC 11889-3. It defines the attributes for each of the commands.

NOTE This file is currently produced by an automated process. Files produced from ISO/IEC 11889-2 or ISO/IEC 11889-3 tables through automated processes are not included in ISO/IEC 11889 so that there is no ambiguity about the information in ISO/IEC 11889-2 or ISO/IEC 11889-3 tables being the normative definition.

```
#include     "CommandAttributeData.c"
```

10.4.3 Command Attribute Functions

10.4.3.1 CommandAuthRole()

This function returns the authorization role required of a handle.

Table 91

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```
AUTH_ROLE
CommandAuthRole(
    TPM_CC      commandCode,        // IN: command code
    UINT32      handleIndex         // IN: handle index (zero based)
    )
{
    if(handleIndex > 1)
        return AUTH_NONE;
    if(handleIndex == 0) {
        ATTRIBUTE_TYPE  properties = s_commandAttributes[commandCode - TPM_CC_FIRST];
        if(properties & HANDLE_1_USER) return AUTH_USER;
        if(properties & HANDLE_1_ADMIN) return AUTH_ADMIN;
        if(properties & HANDLE_1_DUP) return AUTH_DUP;
        return AUTH_NONE;
    }
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & HANDLE_2_USER) return AUTH_USER;
    return AUTH_NONE;
}
```


10.4.3.2 CommandIsImplemented()

This function indicates if a command is implemented.

Table 92

Return Value	Meaning
TRUE	if the command is implemented
FALSE	if the command is not implemented

```
BOOL
CommandIsImplemented(
    TPM_CC           commandCode    // IN: command code
    )
{
    if(commandCode < TPM_CC_FIRST || commandCode > TPM_CC_LAST)
        return FALSE;
    if((s_commandAttributes[commandCode - TPM_CC_FIRST] & IS_IMPLEMENTED))
        return TRUE;
    else
        return FALSE;
}
```

10.4.3.3 CommandGetAttribute()

return a TPMA_CC structure for the given command code

```
TPMA_CC
CommandGetAttribute(
    TPM_CC           commandCode    // IN: command code
    )
{
    UINT32      size = sizeof(s_ccAttr) / sizeof(s_ccAttr[0]);
    UINT32      i;
    for(i = 0; i < size; i++) {
        if(s_ccAttr[i].commandIndex == (UINT16) commandCode)
            return s_ccAttr[i];
    }

    // This function should be called in the way that the command code
    // attribute is available.
    FAIL(FATAL_ERROR_INTERNAL);
}
```

10.4.3.4 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed.

Table 93

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
int
```

```
EncryptSize(
    TPM_CC          commandCode    // IN: commandCode
    )
{
    COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];
    if(ca & ENCRYPT_2)
        return 2;
    if(ca & ENCRYPT_4)
        return 4;
    return 0;
}
```

10.4.3.5 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed.

Table 94

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```
int
DecryptSize(
    TPM_CC          commandCode    // IN: commandCode
    )
{
    COMMAND_ATTRIBUTES  ca = s_commandAttributes[commandCode - TPM_CC_FIRST];

    if(ca & DECRYPT_2)
        return 2;
    if(ca & DECRYPT_4)
        return 4;
    return 0;
}
```

10.4.3.6 IsSessionAllowed()

This function indicates if the command is allowed to have sessions.

This function must not be called if the command is not known to be implemented.

Table 95

Return Value	Meaning
TRUE	session is allowed with this command
FALSE	session is not allowed with this command

```
BOOL
IsSessionAllowed(
    TPM_CC          commandCode    // IN: the command to be checked
    )
{
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & NO_SESSIONS)
        return FALSE;
    else
```

```
        return TRUE;
}
```

10.4.3.7 IsHandleInResponse()

```
BOOL
IsHandleInResponse(
    TPM_CC          commandCode
    )
{
    if(s_commandAttributes[commandCode - TPM_CC_FIRST] & R_HANDLE)
        return TRUE;
    else
        return FALSE;
}
```

10.4.3.8 IsWriteOperation()

Checks to see if an operation will write to NV memory.

```
BOOL
IsWriteOperation(
    TPM_CC          command         // IN: Command to check
    )
{
    switch (command)
    {
        case TPM_CC_NV_Write:
        case TPM_CC_NV_Increment:
        case TPM_CC_NV_SetBits:
        case TPM_CC_NV_Extend:
        // Nv write lock counts as a write operation for authorization purposes.
        // We check to see if the NV is write locked before we do the authorization
        // If it is locked, we fail the command early.
        case TPM_CC_NV_WriteLock:
            return TRUE;
        default:
            break;
    }
    return FALSE;
}
```

10.4.3.9 IsReadOperation()

Checks to see if an operation will write to NV memory.

```
BOOL
IsReadOperation(
    TPM_CC          command         // IN: Command to check
    )
{
    switch (command)
    {
        case TPM_CC_NV_Read:
        case TPM_CC_PolicyNV:
        case TPM_CC_NV_Certify:
        // Nv read lock counts as a read operation for authorization purposes.
        // We check to see if the NV is read locked before we do the authorization
        // If it is locked, we fail the command early.
        case TPM_CC_NV_ReadLock:
            return TRUE;
        default:
```

```
            break;
    }
    return FALSE;
}
```

10.4.3.10 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Table 96

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```
TPMI_YES_NO
CommandCapGetCCList(
    TPM_CC          commandCode,    // IN: start command code
    UINT32          count,          // IN: maximum count for number of entries in
                                    //     'commandList'
    TPML_CCA       *commandList     // OUT: list of TPMA_CC
    )
{
    TPMI_YES_NO      more = NO;
    UINT32           i;

    // initialize output handle list count
    commandList->count = 0;

    // The maximum count of commands that may be return is MAX_CAP_CC.
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;

    // If the command code is smaller than TPM_CC_FIRST, start from TPM_CC_FIRST
    if(commandCode < TPM_CC_FIRST) commandCode = TPM_CC_FIRST;

    // Collect command attributes
    for(i = commandCode; i <= TPM_CC_LAST; i++)
    {
        if(CommandIsImplemented(i))
        {
            if(commandList->count < count)
            {
                // If the list is not full, add the attributes for this command.
                commandList->commandAttributes[commandList->count]
                    = CommandGetAttribute(i);
                commandList->count++;
            }
            else
            {
                // If the list is full but there are more commands to report,
                // indicate this and return.
                more = YES;
                break;
            }
        }
    }
    return more;
}
```

10.5 DRTM.c

10.5.1 Description

This file contains functions that simulate the DRTM events.

10.5.2 Includes

```
#include    "InternalRoutines.h"
```

10.5.3 Functions

10.5.3.1 Signal_Hash_Start()

This function interfaces between the platform code and _TPM_Hash_Start().

```
LIB_EXPORT void
Signal_Hash_Start(
    void
    )
{
    _TPM_Hash_Start();
    return;
}
```

10.5.3.2 Signal_Hash_Data()

This function interfaces between the platform code and _TPM_Hash_Data().

```
LIB_EXPORT void
Signal_Hash_Data(
    unsigned int     size,
    unsigned char   *buffer
    )
{
    _TPM_Hash_Data(size, buffer);
    return;
}
```

10.5.3.3 Signal_Hash_End()

This function interfaces between the platform code and _TPM_Hash_End().

```
LIB_EXPORT void
Signal_Hash_End(
    void
    )
{
    _TPM_Hash_End();
    return;
}
```


10.6 Entity.c

10.6.1 Description

The functions in this file are used for accessing properties for handles of various types. Functions in other files require handles of a specific type but the functions in this file allow use of any handle type.

10.6.2 Includes

```
#include "InternalRoutines.h"
```

10.6.3 Functions

10.6.3.1 EntityGetLoadStatus()

This function will indicate if the entity associated with a handle is present in TPM memory. If the handle is a persistent object handle, and the object exists, the persistent object is moved from NV memory into a RAM object slot and the persistent handle is replaced with the transient object handle for the slot.

Table 97

Error Returns	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_H0	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
TPM_RC
EntityGetLoadStatus(
    TPM_HANDLE       *handle,       // IN/OUT: handle of the entity
    TPM_CC            commandCode   // IN: the commmandCode
    )
{
    TPM_RC           result = TPM_RC_SUCCESS;

    switch(HandleGetType(*handle))
    {
        // For handles associated with hierarchies, the entity is present
        // only if the associated enable is SET.
        case TPM_HT_PERMANENT:
            switch(*handle)
            {
                case TPM_RH_OWNER:
                    if(!gc.shEnable)
                        result = TPM_RC_HIERARCHY;
                    break;

#ifdef  VENDOR_PERMANENT
                case VENDOR_PERMANENT:
#endif
                case TPM_RH_ENDORSEMENT:
                    if(!gc.ehEnable)
                        result = TPM_RC_HIERARCHY;
                    break;
                case TPM_RH_PLATFORM:
                    if(!g_phEnable)
                        result = TPM_RC_HIERARCHY;
```

```
                    break;
                    // null handle, PW session handle and lockout
                    // handle are always available
                case TPM_RH_NULL:
                case TPM_RS_PW:
                case TPM_RH_LOCKOUT:
                    break;
                default:
                    // handling of the manufacture_specific handles
                    if(      ((TPM_RH)*handle >= TPM_RH_AUTH_00)
                        &&  ((TPM_RH)*handle <= TPM_RH_AUTH_FF))
                        // use the value that would have been returned from
                        // unmarshaling if it did the handle filtering
                            result = TPM_RC_VALUE;
                    else
                        pAssert(FALSE);
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // For a transient object, check if the handle is associated
            // with a loaded object.
            if(!ObjectIsPresent(*handle))
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_PERSISTENT:
            // Persistent object
            // Copy the persistent object to RAM and replace the handle with the
            // handle of the assigned slot.  A TPM_RC_OBJECT_MEMORY,
            // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
            // ObjectLoadEvict()
            result = ObjectLoadEvict(handle, commandCode);
            break;
        case TPM_HT_HMAC_SESSION:
            // For an HMAC session, see if the session is loaded
            // and if the session in the session slot is actually
            // an HMAC session.
            if(SessionIsLoaded(*handle))
            {
                SESSION             *session;
                session = SessionGet(*handle);
                // Check if the session is a HMAC session
                if(session->attributes.isPolicy == SET)
                    result = TPM_RC_HANDLE;
            }
            else
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_POLICY_SESSION:
            // For a policy session, see if the session is loaded
            // and if the session in the session slot is actually
            // a policy session.
            if(SessionIsLoaded(*handle))
            {
                SESSION             *session;
                session = SessionGet(*handle);
                // Check if the session is a policy session
                if(session->attributes.isPolicy == CLEAR)
                    result = TPM_RC_HANDLE;
            }
            else
                result = TPM_RC_REFERENCE_H0;
            break;
        case TPM_HT_NV_INDEX:
            // For an NV Index, use the platform-specific routine
            // to search the IN Index space.
```



```
            result = NvIndexIsAccessible(*handle, commandCode);
            break;
        case TPM_HT_PCR:
            // Any PCR handle that is unmarshaled successfully referenced
            // a PCR that is defined.
            break;
        default:
            // Any other handle type is a defect in the unmarshaling code.
            pAssert(FALSE);
            break;
    }
    return result;
}
```

10.6.3.2 EntityGetAuthValue()

This function is used to access the *authValue* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authValue* should have been verified by IsAuthValueAvailable().

This function copies the authorization value of the entity to *auth*.

Return value is the number of octets copied to *auth*.

```
UINT16
EntityGetAuthValue(
    TPMI_DH_ENTITY    handle,        // IN: handle of entity
    AUTH_VALUE       *auth           // OUT: authValue of the entity
    )
{
    TPM2B_AUTH        authValue = {0};

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            switch(handle)
            {
                case TPM_RH_OWNER:
                    // ownerAuth for TPM_RH_OWNER
                    authValue = gp.ownerAuth;
                    break;
                case TPM_RH_ENDORSEMENT:
                    // endorsementAuth for TPM_RH_ENDORSEMENT
                    authValue = gp.endorsementAuth;
                    break;
                case TPM_RH_PLATFORM:
                    // platformAuth for TPM_RH_PLATFORM
                    authValue = gc.platformAuth;
                    break;
                case TPM_RH_LOCKOUT:
                    // lockoutAuth for TPM_RH_LOCKOUT
                    authValue = gp.lockoutAuth;
                    break;
                case TPM_RH_NULL:
                    // nullAuth for TPM_RH_NULL. Return 0 directly here
                    return 0;
                    break;
#ifdef  VENDOR_PERMANENT
                case VENDOR_PERMANENT:
                    // vendor auth value
                    authValue = g_platformUniqueDetails;
#endif
                default:
```

```
                    // If any other permanent handle is present it is
                    // a code defect.
                    pAssert(FALSE);
                    break;
                }
                break;
            case TPM_HT_TRANSIENT:
                // authValue for an object
                // A persistent object would have been copied into RAM
                // and would have an transient object handle here.
                {
                    OBJECT          *object;
                    object = ObjectGet(handle);
                    // special handling if this is a sequence object
                    if(ObjectIsSequence(object))
                    {
                        authValue = ((HASH_OBJECT *)object)->auth;
                    }
                    else
                    {
                        // Auth value is available only when the private portion of
                        // the object is loaded.  The check should be made before
                        // this function is called
                        pAssert(object->attributes.publicOnly == CLEAR);
                        authValue = object->sensitive.authValue;
                    }
                }
                break;
            case TPM_HT_NV_INDEX:
                // authValue for an NV index
                {
                    NV_INDEX        nvIndex;
                    NvGetIndexInfo(handle, &nvIndex);
                    authValue = nvIndex.authValue;
                }
                break;
            case TPM_HT_PCR:
                // authValue for PCR
                PCRGetAuthValue(handle, &authValue);
                break;
            default:
                // If any other handle type is present here, then there is a defect
                // in the unmarshaling code.
                pAssert(FALSE);
                break;
        }

        // Copy the authValue
        pAssert(authValue.t.size <= <K>sizeof(authValue.t.buffer));
        MemoryCopy(auth, authValue.t.buffer, authValue.t.size, sizeof(TPMU_HA));

        return authValue.t.size;
    }
```


10.6.3.3 EntityGetAuthPolicy()

This function is used to access the *authPolicy* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authPolicy* should have been verified by IsAuthPolicyAvailable().

This function copies the authorization policy of the entity to *authPolicy*.

The return value is the hash algorithm for the policy.


```
TPMI_ALG_HASH
EntityGetAuthPolicy(
    TPMI_DH_ENTITY   handle,        // IN: handle of entity
    TPM2B_DIGEST    *authPolicy     // OUT: authPolicy of the entity
    )
{
    TPMI_ALG_HASH       hashAlg = TPM_ALG_NULL;

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            switch(handle)
            {
                case TPM_RH_OWNER:
                    // ownerPolicy for TPM_RH_OWNER
                    *authPolicy = gp.ownerPolicy;
                    hashAlg = gp.ownerAlg;
                    break;
                case TPM_RH_ENDORSEMENT:
                    // endorsementPolicy for TPM_RH_ENDORSEMENT
                    *authPolicy = gp.endorsementPolicy;
                    hashAlg = gp.endorsementAlg;
                    break;
                case TPM_RH_PLATFORM:
                    // platformPolicy for TPM_RH_PLATFORM
                    *authPolicy = gc.platformPolicy;
                    hashAlg = gc.platformAlg;
                    break;
                case TPM_RH_LOCKOUT:
                    // lockoutPolicy for TPM_RH_LOCKOUT
                    *authPolicy = gp.lockoutPolicy;
                    hashAlg = gp.lockoutAlg;
                    break;
                default:
                    // If any other permanent handle is present it is
                    // a code defect.
                    pAssert(FALSE);
                    break;
            }
            break;
        case TPM_HT_TRANSIENT:
            // authPolicy for an object
            {
                OBJECT *object = ObjectGet(handle);
                *authPolicy = object->publicArea.authPolicy;
                hashAlg = object->publicArea.nameAlg;
            }
            break;
        case TPM_HT_NV_INDEX:
            // authPolicy for a NV index
            {
                NV_INDEX        nvIndex;
                NvGetIndexInfo(handle, &nvIndex);
                *authPolicy = nvIndex.publicArea.authPolicy;
                hashAlg = nvIndex.publicArea.nameAlg;
            }
            break;
        case TPM_HT_PCR:
            // authPolicy for a PCR
            hashAlg = PCRGetAuthPolicy(handle, authPolicy);
            break;
        default:
            // If any other handle type is present it is a code defect.
            pAssert(FALSE);
            break;
    }
```

```
    return hashAlg;
}
```

10.6.3.4 EntityGetName()

This function returns the Name associated with a handle. It will set *name* to the Name and return the size of the Name string.

```
UINT16
EntityGetName(
    TPMI_DH_ENTITY   handle,        // IN: handle of entity
    NAME            *name           // OUT: name of entity
    )
{
    UINT16          nameSize;

    switch(HandleGetType(handle))
    {
        case TPM_HT_TRANSIENT:
            // Name for an object
            nameSize = ObjectGetName(handle, name);
            break;
        case TPM_HT_NV_INDEX:
            // Name for a NV index
            nameSize = NvGetName(handle, name);
            break;
        default:
            // For all other types, the handle is the Name
            nameSize = TPM_HANDLE_Marshal(&handle, (BYTE **)&name, NULL);
            break;
    }
    return nameSize;
}
```

10.6.3.5 EntityGetHierarchy()

This function returns the hierarchy handle associated with an entity.

a) A handle that is a hierarchy handle is associated with itself.

b) An NV index belongs to TPM_RH_PLATFORM if TPMA_NV_PLATFORMCREATE, is SET, otherwise it belongs to TPM_RH_OWNER

c) An object handle belongs to its hierarchy. All other handles belong to the platform hierarchy. or an NV Index.

```
TPMI_RH_HIERARCHY
EntityGetHierarchy(
    TPMI_DH_ENTITY   handle         // IN :handle of entity
    )
{
    TPMI_RH_HIERARCHY       hierarcy = TPM_RH_NULL;

    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // hierarchy for a permanent handle
            switch(handle)
            {
                case TPM_RH_PLATFORM:
                case TPM_RH_ENDORSEMENT:
                case TPM_RH_NULL:
                    hierarchy = handle;
```



```
                    break;
                // all other permanent handles are associated with the owner
                // hierarchy. (should only be TPM_RH_OWNER and TMP_RH_LOCKOUT)
                default:
                    hierarcy = TPM_RH_OWNER;
                    break;
            }
            break;
        case TPM_HT_NV_INDEX:
            // hierarchy for NV index
            {
                NV_INDEX        nvIndex;
                NvGetIndexInfo(handle, &nvIndex);
                // If only the platform can delete the index, then it is
                // considered to be in the platform hierarchy, otherwise it
                // is in the owner hierarchy.
                if(nvIndex.publicArea.attributes.TPMA_NV_PLATFORMCREATE == SET)
                    hierarcy = TPM_RH_PLATFORM;
                else
                    hierarcy = TPM_RH_OWNER;
            }
            break;
        case TPM_HT_TRANSIENT:
            // hierarchy for an object
            {
                OBJECT          *object;
                object = ObjectGet(handle);
                if(object->attributes.ppsHierarchy)
                {
                    hierarcy = TPM_RH_PLATFORM;
                }
                else if(object->attributes.epsHierarchy)
                {
                    hierarcy = TPM_RH_ENDORSEMENT;
                }
                else if(object->attributes.spsHierarchy)
                {
                    hierarcy = TPM_RH_OWNER;
                }

            }
            break;
        case TPM_HT_PCR:
            hierarcy = TPM_RH_OWNER;
            break;
        default:
            pAssert(0);
            break;
    }
    // this is unreachable but it provides a return value for the default
    // case which makes the complier happy
    return hierarcy;
}
```

10.7 Global.c

10.7.1 Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables is in Global.h.

10.7.2 Includes and Defines

```
#define GLOBAL_C
#include "InternalRoutines.h"
```

10.7.3 Global Data Values

These values are visible across multiple modules.

```
BOOL                    g_phEnable;
const UINT16            g_rcIndex[15] = {TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,
                                          TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,
                                          TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,
                                          TPM_RC_D, TPM_RC_E, TPM_RC_F
                                         };
TPM_HANDLE              g_exclusiveAuditSession;
UINT64                  g_time;
BOOL                    g_pcrReConfig;
TPMI_DH_OBJECT          g_DRTMHandle;
BOOL                    g_DrtmPreStartup;
BOOL                    g_clearOrderly;
TPM_SU                  g_prevOrderlyState;
BOOL                    g_updateNV;
BOOL                    g_nvOk;
TPM2B_AUTH              g_platformUniqueDetails;
STATE_CLEAR_DATA        gc;
STATE_RESET_DATA        gr;
PERSISTENT_DATA         gp;
ORDERLY_DATA            go;
```

10.7.4 Private Values

10.7.4.1 SessionProcess.c

```
#ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
```

These values do not need to be retained between commands.

```
TPM_HANDLE              s_sessionHandles[MAX_SESSION_NUM];
TPMA_SESSION            s_attributes[MAX_SESSION_NUM];
TPM_HANDLE              s_associatedHandles[MAX_SESSION_NUM];
TPM2B_NONCE             s_nonceCaller[MAX_SESSION_NUM];
TPM2B_AUTH              s_inputAuthValues[MAX_SESSION_NUM];
UINT32                  s_encryptSessionIndex;
UINT32                  s_decryptSessionIndex;
UINT32                  s_auditSessionIndex;
TPM2B_DIGEST            s_cpHashForAudit;
UINT32                  s_sessionNum;
#endif  // __IGNORE_STATE__
BOOL                    s_DAPendingOnNV;
#ifdef TPM_CC_GetCommandAuditDigest
TPM2B_DIGEST            s_cpHashForCommandAudit;
#endif
```

10.7.4.2 DA.c

```
UINT64                  s_selfHealTimer;
UINT64                  s_lockoutTimer;
```


10.7.4.3 NV.c

```
UINT32                s_reservedAddr[NV_RESERVE_LAST];
UINT32                s_reservedSize[NV_RESERVE_LAST];
UINT32                s_ramIndexSize;
BYTE                  s_ramIndex[RAM_INDEX_SPACE];
UINT32                s_ramIndexSizeAddr;
UINT32                s_ramIndexAddr;
UINT32                s_maxCountAddr;
UINT32                s_evictNvStart;
UINT32                s_evictNvEnd;
TPM_RC                s_NvStatus;
```

10.7.4.4 Object.c

```
OBJECT_SLOT           s_objects[MAX_LOADED_OBJECTS];
```

10.7.4.5 PCR.c

```
PCR                   s_pcrs[IMPLEMENTATION_PCR];
```

10.7.4.6 Session.c

```
SESSION_SLOT          s_sessions[MAX_LOADED_SESSIONS];
UINT32                s_oldestSavedSession;
int                   s_freeSessionSlots;
```

10.7.4.7 Manufacture.c

```
BOOL                  g_manufactured = FALSE;
```

10.7.4.8 Power.c

```
BOOL                  s_initialized = FALSE;
```

10.7.4.9 MemoryLib.c

The *s_actionOutputBuffer* should not be modifiable by the host system until the TPM has returned a response code. The *s_actionOutputBuffer* should not be accessible until response parameter encryption, if any, is complete. This memory is not used between commands.

```
#ifndef __IGNORE_STATE__        // DO NOT DEFINE THIS VALUE
UINT32   s_actionInputBuffer[1024];          // action input buffer
UINT32   s_actionOutputBuffer[1024];         // action output buffer
BYTE     s_responseBuffer[MAX_RESPONSE_SIZE];// response buffer
#endif
```

10.7.4.10 SelfTest.c

Define these values here if the AlgorithmTests() project is not used

```
#ifndef SELF_TEST
ALGORITHM_VECTOR      g_implementedAlgorithms;
ALGORITHM_VECTOR      g_toTest;
#endif
```

10.7.4.11 TpmFail.c

```
jmp_buf                 g_jumpBuffer;
BOOL                    g_forceFailureMode;
BOOL                    g_inFailureMode;
UINT32                  s_failFunction;
UINT32                  s_failLine;
UINT32                  s_failCode;
```

10.8 Handle.c

10.8.1 Description

This file contains the functions that return the type of a handle.

10.8.2 Includes

```
#include "Tpm.h"
#include "InternalRoutines.h"
```

10.8.3 Functions

10.8.3.1 HandleGetType()

This function returns the type of a handle which is the MSO of the handle.

```
TPM_HT
HandleGetType(
    TPM_HANDLE      handle          // IN: a handle to be checked
    )
{
    // return the upper bytes of input data
    return (TPM_HT) ((handle & HR_RANGE_MASK) >> HR_SHIFT);
}
```

10.8.3.2 NextPermanentHandle()

This function returns the permanent handle that is equal to the input value or is the next higher value. If there is no handle with the input value and there is no next higher value, it returns 0:

```
TPM_HANDLE
NextPermanentHandle(
    TPM_HANDLE      inHandle        // IN: the handle to check
    )
{
    TPM_HANDLE     retVal = 0;

    switch (inHandle)
    {
        case TPM_RH_OWNER:
        case TPM_RH_NULL:
        case TPM_RS_PW:
        case TPM_RH_LOCKOUT:
        case TPM_RH_ENDORSEMENT:
        case TPM_RH_PLATFORM:
        case TPM_RH_PLATFORM_NV:
#ifdef  VENDOR_PERMANENT
        case VENDOR_PERMANENT:
```

```
#endif
            retVal = inHandle;
            break;
        default:
            break;
    }
    return 0;
}
```

10.8.3.3 PermanentCapGetHandles()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

Table 98

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```
TPMI_YES_NO
PermanentCapGetHandles(
    TPM_HANDLE       handle,        // IN: start handle
    UINT32           count,         // IN: count of returned handles
    TPML_HANDLE     *handleList     // OUT: list of handle
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate permanent handle range
    for(i = NextPermanentHandle(handle);
            i != 0; i = NextPermanentHandle(i+1))
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this permanent
            // handle to it
            handleList->handle[handleList->count] = i;
            handleList->count++;
        }
        else
        {
            // If the return list is full but we still have permanent handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

10.9 Locality.c

10.9.1 Includes

```
#include "InternalRoutines.h"
```

10.9.2 LocalityGetAttributes()

This function will convert a locality expressed as an integer into TPMA_LOCALITY form.

The function returns the locality attribute.

```
TPMA_LOCALITY
LocalityGetAttributes(
    UINT8           locality       // IN: locality value
    )
{
    TPMA_LOCALITY             locality_attributes;
    BYTE                     *localityAsByte = (BYTE *)&locality_attributes;

    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
    switch(locality)
    {
        case 0:
            locality_attributes.TPM_LOC_ZERO = SET;
            break;
        case 1:
            locality_attributes.TPM_LOC_ONE = SET;
            break;
        case 2:
            locality_attributes.TPM_LOC_TWO = SET;
            break;
        case 3:
            locality_attributes.TPM_LOC_THREE = SET;
            break;
        case 4:
            locality_attributes.TPM_LOC_FOUR = SET;
            break;
        default:
            pAssert(locality < 256 && locality > 31);
            *localityAsByte = locality;
            break;
    }
    return locality_attributes;
}
```

10.10 Manufacture.c

10.10.1 Description

This file contains the function that performs the **manufacturing** of the TPM in a simulated environment. These functions should not be used outside of a manufacturing or simulation environment.

10.10.2 Includes and Data Definitions

```
#define MANUFACTURE_C
#include "InternalRoutines.h"
#include "Global.h"
```


10.10.3 Functions

10.10.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be re-manufactured by calling TPM_Teardown() first and then calling this function again.

Table 99

Return Value	Meaning
0	success
1	manufacturing process previously performed

```
LIB_EXPORT int
TPM_Manufacture(
    BOOL                firstTime       // IN: indicates if this is the first call from
                                        //     main()
    )
{
    TPM_SU              orderlyShutdown;
    UINT64              totalResetCount = 0;

    // If TPM has been manufactured, return indication.
    if(!firstTime && g_manufactured)
        return 1;

    // initialize crypto units
    //CryptInitUnits();

    //
    s_selfHealTimer = 0;
    s_lockoutTimer = 0;
    s_DAPendingOnNV = FALSE;


    // initialize NV
    NvInit();

#ifdef _DRBG_STATE_SAVE
    // Initialize the drbg. This needs to come before the install
    // of the hierarchies
    if(!_cpri__Startup())                   // Have to start the crypto units first
        FAIL(FATAL_ERROR_INTERNAL);
    _cpri__DrbgGetPutState(PUT_STATE, 0, NULL);
#endif

    // default configuration for PCR
    PCRSimStart();

    // initialize pre-installed hierarchy data
    // This should happen after NV is initialized because hierarchy data is
    // stored in NV.
    HierarchyPreInstall_Init();

    // initialize dictionary attack parameters
    DAPreInstall_Init();

    // initialize PP list
    PhysicalPresencePreInstall_Init();

```

```
    // initialize command audit list
    CommandAuditPreInstall_Init();

    // first start up is required to be Startup(CLEAR)
    orderlyShutdown = TPM_SU_CLEAR;
    NvWriteReserved(NV_ORDERLY, &orderlyShutdown);

    // initialize the firmware version
    gp.firmwareV1 = FIRMWARE_V1;
#ifdef FIRMWARE_V2
    gp.firmwareV2 = FIRMWARE_V2;
#else
    gp.firmwareV2 = 0;
#endif
    NvWriteReserved(NV_FIRMWARE_V1, &gp.firmwareV1);
    NvWriteReserved(NV_FIRMWARE_V2, &gp.firmwareV2);

    // initialize the total reset counter to 0
    NvWriteReserved(NV_TOTAL_RESET_COUNT, &totalResetCount);

    // initialize the clock stuff
    go.clock = 0;
    go.clockSafe = YES;

#ifdef _DRBG_STATE_SAVE
    // initialize the current DRBG state in NV

    _cpri__DrbgGetPutState(GET_STATE, sizeof(go.drbgState), (BYTE *)&go.drbgState);
#endif

    NvWriteReserved(NV_ORDERLY_DATA, &go);

    // Commit NV writes.  Manufacture process is an artificial process existing
    // only in simulator environment and it is not defined in ISO/IEC 11889
    // what should be the expected behavior if the NV write fails at this
    // point.  Therefore, it is assumed the NV write here is always success and
    // no return code of this function is checked.
    NvCommit();

    g_manufactured = TRUE;

    return 0;
}
```


10.10.3.2 TPM_TearDown()

This function prepares the TPM for re-manufacture. It should not be implemented in anything other than a simulated TPM.

In this implementation, all that is needs is to stop the cryptographic units and set a flag to indicate that the TPM can be re-manufactured. This should be all that is necessary to start the manufacturing process again.

Table 100

Return Value	Meaning
0	success
1	TPM not previously manufactured

```
LIB_EXPORT int
TPM_TearDown(
    void
```



```
    )
{
    // stop crypt units
    CryptStopUnits();

    g_manufactured = FALSE;
    return 0;
}
```

10.11 Marshal.c

10.11.1 Introduction

This file contains the marshaling and unmarshaling code.

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets ("<>") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

10.11.2 Unmarshal and Marshal a Value

In ISO/IEC 11889-2, Table 39, the TPMI_DH_OBJECT type is defined as shown in Table 101:

Table 101— Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type from ISO/IEC 11889-2

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```
TPM_RC
TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
                         bool flag)
{
    TPM_RC     result;
    result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if (*target == TPM_RH_NULL) {
        if(flag)
            return TPM_RC_SUCCESS;
        else
            return TPM_RC_VALUE;
    }
    if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
        if((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
            return TPM_RC_VALUE;
    return TPM_RC_SUCCESS;
}
```

and the following marshaling code:

NOTE 1 The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.

```
UINT16
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}
```

10.11.3 Unmarshal and Marshal a Union

In ISO/IEC 11889-2, Table 185, the TPMU_PUBLIC_PARMS union is defined as shown in Table 102:

Table 102 — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S> from ISO/IEC 11889-2

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign | encrypt | neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys

From this table, the following unmarshaling code is generated:

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
                             UINT32 selector)
{
    switch(selector) {
#ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                          (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
#endif
#ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal(
                    (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
#endif
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
#endif
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
#endif
    }
    return TPM_RC_SELECTOR;
}
```


NOTE 2 The **#ifdef/#endif** directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
UINT16
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
                           UINT32 selector)
{
    switch(selector) {
#ifdef TPM_ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Marshal(
                          (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
#endif
#ifdef TPM_ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Marshal(
                           (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
#endif
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Marshal(
                            (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
#endif
#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Marshal(
                             (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
#endif
    }
    assert(1);
    return 0;
}
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The next clause illustrates this.

10.11.4 Unmarshal and Marshal a Structure

In ISO/IEC 11889-2, Table 187, the TPMT_PUBLIC structure is defined as shown in Table 103:

Table 103 — Definition of TPMT_PUBLIC Structure from ISO/IEC 11889-2

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object
objectAttributes	TPMA_OBJECT	attributes that, along with *type*, determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the *nameAlg* of the object.
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```
TPM_RC
TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, bool flag)
{
    TPM_RC    result;
    result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
                                       buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
                                     buffer, size, flag);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
                                   buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
                                    buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
                                         buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
                                      buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```


The marshaling code for the TPMT_PUBLIC structure is:

```
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
                               (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
                              (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
;
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                       (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));

    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
                            (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));

    result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
                       (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
                                                         (UINT32)source->type));

    result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
                              (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
                                                         (UINT32)source->type));

    return result;
}
```

10.11.5 Unmarshal and Marshal an Array

In ISO/IEC 11889-2, Table 98, the TPML_DIGEST Structure is defined as shown in Table 104:

Table 104 — Definition of TPML_DIGEST Structure from ISO/IEC 11889-2

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```
TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2))    // This check is triggered by the {2:} notation
                                // on 'count'
        return TPM_RC_SIZE;

    if((target->count) > 8)     // This check is triggered by the {:8} notation
                                // on 'digests'.
        return TPM_RC_SIZE;

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *)(target->digests),
                                          buffer, size, (INT32)(target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```
UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16)(result + UINT32_Marshal((UINT32 *)&(source->count), buffer,
                                                                            size));
    result = (UINT16)(result + TPM2B_DIGEST_Array_Marshal(
                                (TPM2B_DIGEST *)(source->digests), buffer, size,
                                (INT32)(source->count)));

    return result;
}
```

The marshaling code for the array is:

```
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                              INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```

10.11.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 10.11.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

In ISO/IEC 11889-2, Table 76, the TPM2B_EVENT is defined as shown in Table 105:

Table 105 — Definition of TPM2B_EVENT Structure from ISO/IEC 11889-2

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand


```
TPM_RC
TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    // if size equal to 0, the rest of the structure is a zero buffer.  Stop
 processing
    if(target->t.size == 0)
        return TPM_RC_SUCCESS;

    if((target->t.size) > 1024)    // This check is triggered by the {:1024} notation
                                   // on 'buffer'

        return TPM_RC_SIZE;

    result = BYTE_Array_Unmarshal((BYTE *)(target->t.buffer), buffer, size,
                                  (INT32)(target->t.size));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}
```

Which use these structure definitions:

```
typedef struct {
    UINT16          size;
    BYTE            buffer[1];
} TPM2B;

typedef struct {
    UINT16          size;
    BYTE            buffer[1024];
} EVENT_2B;

typedef union {
    EVENT_2B        t;     // The type-specific union member
    TPM2B           b;     // The generic union member
} TPM2B_EVENT;
```

10.12 MemoryLib.c

10.12.1 Description

This file contains a set of miscellaneous memory manipulation routines. Many of the functions have the same semantics as functions defined in string.h. Those functions are not used in the TPM in order to avoid namespace contamination.

10.12.2 Includes and Data Definitions

```
#define MEMORY_LIB_C
#include "InternalRoutines.h"
```

These buffers are set aside to hold command and response values. In this implementation, it is not guaranteed that the code will stop accessing the *s_actionInputBuffer* before starting to put values in the *s_actionOutputBuffer* so different buffers are required. However, the *s_actionInputBuffer* and *s_responseBuffer* are not needed at the same time and they could be the same buffer.

10.12.3 Functions on BYTE Arrays

10.12.3.1 MemoryMove()

This function moves data from one place in memory to another. No safety checks of any type are performed. If source and data buffer overlap, then the move is done as if an intermediate buffer were used.

NOTE This function is used by MemoryCopy(), MemoryCopy2B(), and MemoryConcat2b() and needs the caller to know the maximum size of the destination buffer so that there is no possibility of buffer overrun.

```
LIB_EXPORT void
MemoryMove(
    void              *destination,   // OUT: move destination
    const void        *source,        // IN: move source
    UINT32             size,          // IN: number of octets to moved
    UINT32             dSize          // IN: size of the receive buffer
    )
{
    const BYTE *p = (BYTE *)source;
    BYTE *q = (BYTE *)destination;

    if(destination == NULL || source == NULL)
        return;

    pAssert(size <= dSize);
    // if the destination buffer has a lower address than the
    // source, then moving bytes in ascending order is safe.
    dSize -= size;

    if (p>q || (p+size <= q))
    {
        while(size--)
            *q++ = *p++;
    }
    // If the destination buffer has a higher address than the
    // source, then move bytes from the end to the beginning.
    else if (p < q)
    {
        p += size;
        q += size;
```

```
        while (size--)
            *--q = *--p;
    }

    // If the source and destination address are the same, nothing to move.
    return;
}
```

10.12.3.2 MemoryCopy()

This function moves data from one place in memory to another. No safety checks of any type are performed. If the destination and source overlap, then the results are unpredictable.

```

#ifndef MemoryMove //%
void
MemoryCopy(
    void                *destination,    // OUT: copy destination
    void                *source,         // IN: copy source
    UINT32               size,           // IN: number of octets being copied
    UINT32               dSize           // IN: size of the receive buffer
    )
{
    MemoryMove(destination, source, size, dSize);
}
#else //%
//%#define MemoryCopy(destination, source, size, destSize)        \
//%    MemoryMove((destination), (source), (size), (destSize))
#endif //%
```

10.12.3.3 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Table 106

Return Value	Meaning
TRUE	all octets are the same
FALSE	all octets are not the same

```
LIB_EXPORT BOOL
MemoryEqual(
    const void          *buffer1,        // IN: compare buffer1
    const void          *buffer2,        // IN: compare buffer2
    UINT32               size            // IN: size of bytes being compared
    )
{
    BOOL            equal = TRUE;
    const BYTE     *b1, *b2;

    b1 = (BYTE *)buffer1;
    b2 = (BYTE *)buffer2;

    // Compare all bytes so that there is no leakage of information
    // due to timing differences.
    for(; size > 0; size--)
        equal = (*b1++ == *b2++) && equal;

    return equal;
}
```


10.12.3.4 MemoryCopy2B()

This function copies a TPM2B. This can be used when the TPM2B types are the same or different. No size checking is done on the destination so the caller should make sure that the destination is large enough.

This function returns the number of octets in the data buffer of the TPM2B.

```
LIB_EXPORT INT16
MemoryCopy2B(
    TPM2B           *dest,          // OUT: receiving TPM2B
    const TPM2B     *source,        // IN: source TPM2B
    UINT16           dSize          // IN: size of the receiving buffer
    )
{

    if(dest == NULL)
        return 0;
    if(source == NULL)
        dest->size = 0;
    else
    {
        dest->size = source->size;
        MemoryMove(dest->buffer, source->buffer, dest->size, dSize);
    }
    return dest->size;
}
```

10.12.3.5 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to an the buffer contents of another TPM2B and adjust the size accordingly *(a* := *(a* | *b*)).

```
LIB_EXPORT void
MemoryConcat2B(
    TPM2B           *aInOut,        // IN/OUT: destination 2B
    TPM2B           *bIn,           // IN: second 2B
    UINT16           aSize          // IN: The size of aInOut.buffer (max values for
                                    //     aInOut.size)
    )
{
    MemoryMove(&aInOut->buffer[aInOut->size],
               bIn->buffer,
               bIn->size,
               aSize - aInOut->size);
    aInOut->size = aInOut->size + bIn->size;
    return;
}
```

10.12.3.6 Memory2BEqual()

This function will compare two TPM2B structures. To be equal, they need to be the same size and the buffer contexts need to be the same in all octets.

Table 107

Return Value	Meaning
TRUE	size and buffer contents are the same
FALSE	size or buffer contents are not the same

```
LIB_EXPORT BOOL
Memory2BEqual(
    const TPM2B      *aIn,            // IN: compare value
    const TPM2B      *bIn             // IN: compare value
    )
{
    if(aIn->size != bIn->size)
        return FALSE;

    return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
}
```

10.12.3.7 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE The **dSize** parameter forces the caller to know how big the receiving buffer is to make sure that there is no possibility that the caller will inadvertently run over the end of the buffer.

```
LIB_EXPORT void
MemorySet(
    void            *destination,    // OUT: memory destination
    char             value,          // IN: fill value
    UINT32           size            // IN: number of octets to fill
    )
{
    char *p = (char *)destination;
    while (size--)
        *p++ = value;
    return;
}
```

10.12.3.8 MemoryGetActionInputBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
BYTE *
MemoryGetActionInputBuffer(
    UINT32           size            // Size, in bytes, required for the input
                                     // unmarshaling
    )
{
    BYTE        *buf = NULL;

    if(size > 0)
    {
        // In this implementation, a static buffer is set aside for action output.
        // Other implementations may apply additional optimization based on command
        // code or other factors.
        UINT32      *p = s_actionInputBuffer;
        buf = (BYTE *)p;
        pAssert(size < <K>sizeof(s_actionInputBuffer));

        // size of an element in the buffer
```



```
#define SZ      sizeof(s_actionInputBuffer[0])

        for(size = (size + SZ - 1) / SZ; size > 0; size--)
            *p++ = 0;
#undef SZ
    }
    return buf;
}
```

10.12.3.9 MemoryGetActionOutputBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```
void *
MemoryGetActionOutputBuffer(
    TPM_CC              command         // Command that requires the buffer
    )
{
    // In this implementation, a static buffer is set aside for action output.
    // Other implementations may apply additional optimization based on the command
    // code or other factors.
    command = 0;        // Unreferenced parameter
    return s_actionOutputBuffer;
}
```

10.12.3.10 MemoryGetResponseBuffer()

This function returns the address into which the command response is marshaled from values in the action output buffer.

```
BYTE *
MemoryGetResponseBuffer(
    TPM_CC              command         // Command that requires the buffer
    )
{
    // In this implementation, a static buffer is set aside for responses.
    // Other implementation may apply additional optimization based on the command
    // code or other factors.
    command = 0;        // Unreferenced parameter
    return s_responseBuffer;
}
```

10.12.3.11 MemoryRemoveTrailingZeros()

This function is used to adjust the length of an authorization value. It adjusts the size of the TPM2B so that it does not include octets at the end of the buffer that contain zero. The function returns the number of non-zero octets in the buffer.

```
UINT16
MemoryRemoveTrailingZeros (
    TPM2B_AUTH      *auth           // IN/OUT: value to adjust
    )
{
    BYTE        *a = &auth->t.buffer[auth->t.size-1];
    for(; auth->t.size > 0; auth->t.size--)
    {
        if(*a--)
            break;
    }
    return auth->t.size;
```

```
}
```

10.13 Power.c

10.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

10.13.2 Includes and Data Definitions

```
#define POWER_C
#include "InternalRoutines.h"
```

10.13.3 Functions

10.13.3.1 TPMInit()

This function is used to process a power on event.

```
void
TPMInit(
    void
    )
{
    // Set state as not initialized. This means that Startup is required
    s_initialized = FALSE;

    return;
}
```

10.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
void
TPMRegisterStartup(
    void
    )
{
    s_initialized = TRUE;

    return;
}
```

10.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

Table 108

Return Value	Meaning
TRUE	TPM has been initialized
FALSE	TPM has not been initialized

```
BOOL
TPMIsStarted(
    void
    )
{
    return s_initialized;
}
```

10.14 PropertyCap.c

10.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

10.14.2 Includes

```
#include "InternalRoutines.h"
```

10.14.3 Functions

10.14.3.1 PCRGetProperty()

This function accepts a property selection and, if so, sets *value* to the value of the property.

All the fixed values are vendor dependent or determined by a platform-specific specification. The values in the table below are examples and should be changed by the vendor.

Table 109

Return Value	Meaning
TRUE	referenced property exists and *value* set
FALSE	referenced property does not exist

```
static BOOL
TPMPropertyIsDefined(
    TPM_PT              property,       // IN: property
    UINT32             *value           // OUT: property value
    )
{
    switch(property)
    {
        case TPM_PT_FAMILY_INDICATOR:
            // from the title page of ISO/IEC 11889
            // For ISO/IEC 11889, the value is "2.0".
            *value = TPM_SPEC_FAMILY;
            break;
        case TPM_PT_LEVEL:
            // from the title page of ISO/IEC 11889
            *value = TPM_SPEC_LEVEL;
            break;
```

```
        case TPM_PT_REVISION:
            // from the title page of ISO/IEC 11889
            *value = TPM_SPEC_VERSION;
            break;
        case TPM_PT_DAY_OF_YEAR:
            // computed from the date value on the title page of ISO/IEC 11889
            *value = TPM_SPEC_DAY_OF_YEAR;
            break;
        case TPM_PT_YEAR:
            // from the title page of ISO/IEC 11889
            *value = TPM_SPEC_YEAR;
            break;
        case TPM_PT_MANUFACTURER:
            // vendor ID unique to each TPM manufacturer
            *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
            break;
        case TPM_PT_VENDOR_STRING_1:
            // first four characters of the vendor ID string
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
            break;
        case TPM_PT_VENDOR_STRING_2:
            // second four characters of the vendor ID string
#ifdef VENDOR_STRING_2
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
#else
            *value = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_3:
            // third four characters of the vendor ID string
#ifdef VENDOR_STRING_3
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
#else
            *value = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_4:
            // fourth four characters of the vendor ID string
#ifdef VENDOR_STRING_4
            *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
#else
            *value = 0;
#endif
            break;
        case TPM_PT_VENDOR_TPM_TYPE:
            // vendor-defined value indicating the TPM model
            *value = 1;
            break;
        case TPM_PT_FIRMWARE_VERSION_1:
            // more significant 32-bits of a vendor-specific value
            *value = gp.firmwareV1;
            break;
        case TPM_PT_FIRMWARE_VERSION_2:
            // less significant 32-bits of a vendor-specific value
            *value = gp.firmwareV2;
            break;
        case TPM_PT_INPUT_BUFFER:
            // maximum size of TPM2B_MAX_BUFFER
            *value = MAX_DIGEST_BUFFER;
            break;
        case TPM_PT_HR_TRANSIENT_MIN:
            // minimum number of transient objects that can be held in TPM
            // RAM
            *value = MAX_LOADED_OBJECTS;
            break;
        case TPM_PT_HR_PERSISTENT_MIN:
```

```
        // minimum number of persistent objects that can be held in
        // TPM NV memory
        // In this implementation, there is no minimum number of
        // persistent objects.
        *value = MIN_EVICT_OBJECTS;
        break;
    case TPM_PT_HR_LOADED_MIN:
        // minimum number of authorization sessions that can be held in
        // TPM RAM
        *value = MAX_LOADED_SESSIONS;
        break;
    case TPM_PT_ACTIVE_SESSIONS_MAX:
        // number of authorization sessions that may be active at a time
        *value = MAX_ACTIVE_SESSIONS;
        break;
    case TPM_PT_PCR_COUNT:
        // number of PCR implemented
        *value = IMPLEMENTATION_PCR;
        break;
    case TPM_PT_PCR_SELECT_MIN:
        // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
        *value = PCR_SELECT_MIN;
        break;
    case TPM_PT_CONTEXT_GAP_MAX:
        // maximum allowed difference (unsigned) between the contextID
        // values of two saved session contexts
        *value = (1 << (<K>sizeof(CONTEXT_SLOT) * 8)) - 1;
        break;
    case TPM_PT_NV_COUNTERS_MAX:
        // maximum number of NV indexes that are allowed to have the
        // TPMA_NV_COUNTER attribute SET
        // In this implementation, there is no limitation on the number
        // of counters, except for the size of the NV Index memory.
        *value = 0;
        break;
    case TPM_PT_NV_INDEX_MAX:
        // maximum size of an NV index data area
        *value = MAX_NV_INDEX_SIZE;
        break;
    case TPM_PT_MEMORY:
        // a TPMA_MEMORY indicating the memory management method for the TPM
    {
        TPMA_MEMORY         attributes = {0};
        attributes.sharedNV = SET;
        attributes.objectCopiedToRam = SET;

        // Note: Different compilers may require a different method to cast
        // a bit field structure to a UINT32.
        *value = * (UINT32 *) &attributes;
        break;
    }
    case TPM_PT_CLOCK_UPDATE:
        // interval, in seconds, between updates to the copy of
        // TPMS_TIME_INFO .clock in NV
        *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
        break;
    case TPM_PT_CONTEXT_HASH:
        // algorithm used for the integrity hash on saved contexts and
        // for digesting the fuData of TPM2_FirmwareRead()
        *value = CONTEXT_INTEGRITY_HASH_ALG;
        break;
    case TPM_PT_CONTEXT_SYM:
        // algorithm used for encryption of saved contexts
        *value = CONTEXT_ENCRYPT_ALG;
        break;
    case TPM_PT_CONTEXT_SYM_SIZE:
```

```
            // size of the key used for encryption of saved contexts
            *value = CONTEXT_ENCRYPT_KEY_BITS;
            break;
        case TPM_PT_ORDERLY_COUNT:
            // maximum difference between the volatile and non-volatile
            // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
            *value = MAX_ORDERLY_COUNT;
            break;
        case TPM_PT_MAX_COMMAND_SIZE:
            // maximum value for 'commandSize'
            *value = MAX_COMMAND_SIZE;
            break;
        case TPM_PT_MAX_RESPONSE_SIZE:
            // maximum value for 'responseSize'
            *value = MAX_RESPONSE_SIZE;
            break;
        case TPM_PT_MAX_DIGEST:
            // maximum size of a digest that can be produced by the TPM
            *value = sizeof(TPMU_HA);
            break;
        case TPM_PT_MAX_OBJECT_CONTEXT:
            // maximum size of a TPMS_CONTEXT that will be returned by
            // TPM2_ContextSave for object context
            *value = 0;

            // adding sequence, saved handle and hierarchy
            *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
                      sizeof(TPMI_RH_HIERARCHY);
            // add size field in TPM2B_CONTEXT
            *value += sizeof(UINT16);

            // add integrity hash size
            *value += sizeof(UINT16) +
                      CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);

            // Add fingerprint size, which is the same as sequence size
            *value += sizeof(UINT64);

            // Add OBJECT structure size
            *value += sizeof(OBJECT);
            break;
        case TPM_PT_MAX_SESSION_CONTEXT:
            // the maximum size of a TPMS_CONTEXT that will be returned by
            // TPM2_ContextSave for object context
            *value = 0;

            // adding sequence, saved handle and hierarchy
            *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
                      sizeof(TPMI_RH_HIERARCHY);
            // Add size field in TPM2B_CONTEXT
            *value += sizeof(UINT16);

            // Add integrity hash size
            *value += sizeof(UINT16) +
                      CryptGetHashDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
            // Add fingerprint size, which is the same as sequence size
            *value += sizeof(UINT64);

            // Add SESSION structure size
            *value += sizeof(SESSION);
            break;
        case TPM_PT_PS_FAMILY_INDICATOR:
            // platform specific values for the TPM_PT_PS parameters from
            // the relevant platform-specific specification
            // In the reference implementation, all of these values are 0.
            *value = 0;
```



```
            break;
    case TPM_PT_PS_LEVEL:
        // level of the platform-specific specification
        *value = 0;
        break;
    case TPM_PT_PS_REVISION:
        // specification Revision times 100 for the platform-specific
        // specification
        *value = 0;
        break;
    case TPM_PT_PS_DAY_OF_YEAR:
        // platform-specific specification day of year using TCG calendar
        *value = 0;
        break;
    case TPM_PT_PS_YEAR:
        // platform-specific specification year using the CE
        *value = 0;
        break;
    case TPM_PT_SPLIT_MAX:
        // number of split signing operations supported by the TPM
        *value = 0;
#ifdef TPM_ALG_ECC
        *value = sizeof(gr.commitArray) * 8;
#endif
        break;
    case TPM_PT_TOTAL_COMMANDS:
        // total number of commands implemented in the TPM
        // Since the reference implementation does not have any
        // vendor-defined commands, this will be the same as the
        // number of library commands.
    {
        UINT32 i;
        *value = 0;

        // calculate implemented command numbers
        for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
        {
            if(CommandIsImplemented(i)) (*value)++;
        }
        break;
    }
    case TPM_PT_LIBRARY_COMMANDS:
        // number of commands from the TPM library that are implemented
    {
        UINT32 i;
        *value = 0;

        // calculate implemented command numbers
        for(i = TPM_CC_FIRST; i <= TPM_CC_LAST; i++)
        {
            if(CommandIsImplemented(i)) (*value)++;
        }
        break;
    }
    case TPM_PT_VENDOR_COMMANDS:
        // number of vendor commands that are implemented
        *value = 0;
        break;
    case TPM_PT_PERMANENT:
        // TPMA_PERMANENT
    {
        TPMA_PERMANENT          flags = {0};
        if(gp.ownerAuth.t.size != 0)
            flags.ownerAuthSet = SET;
        if(gp.endorsementAuth.t.size != 0)
            flags.endorsementAuthSet = SET;
```

```
            if(gp.lockoutAuth.t.size != 0)
                flags.lockoutAuthSet = SET;
            if(gp.disableClear)
                flags.disableClear = SET;
            if(gp.failedTries >= gp.maxTries)
                flags.inLockout = SET;
            // In this implementation, EPS is always generated by TPM
            flags.tpmGeneratedEPS = SET;

            // Note: Different compilers may require a different method to cast
            // a bit field structure to a UINT32.
            *value = * (UINT32 *) &flags;
            break;
        }
        case TPM_PT_STARTUP_CLEAR:
            // TPMA_STARTUP_CLEAR
        {
            TPMA_STARTUP_CLEAR      flags = {0};
            if(g_phEnable)
                flags.phEnable = SET;
            if(gc.shEnable)
                flags.shEnable = SET;
            if(gc.ehEnable)
                flags.ehEnable = SET;
            if(gc.phEnableNV)
                flags.phEnableNV = SET;
            if(g_prevOrderlyState != SHUTDOWN_NONE)
                flags.orderly = SET;

            // Note: Different compilers may require a different method to cast
            // a bit field structure to a UINT32.
            *value = * (UINT32 *) &flags;
            break;
        }
        case TPM_PT_HR_NV_INDEX:
            // number of NV indexes currently defined
            *value = NvCapGetIndexNumber();
            break;
        case TPM_PT_HR_LOADED:
            // number of authorization sessions currently loaded into TPM
            // RAM
            *value = SessionCapGetLoadedNumber();
            break;
        case TPM_PT_HR_LOADED_AVAIL:
            // number of additional authorization sessions, of any type,
            // that could be loaded into TPM RAM
            *value = SessionCapGetLoadedAvail();
            break;
        case TPM_PT_HR_ACTIVE:
            // number of active authorization sessions currently being
            // tracked by the TPM
            *value = SessionCapGetActiveNumber();
            break;
        case TPM_PT_HR_ACTIVE_AVAIL:
            // number of additional authorization sessions, of any type,
            // that could be created
            *value = SessionCapGetActiveAvail();
            break;
        case TPM_PT_HR_TRANSIENT_AVAIL:
            // estimate of the number of additional transient objects that
            // could be loaded into TPM RAM
            *value = ObjectCapGetTransientAvail();
            break;
        case TPM_PT_HR_PERSISTENT:
            // number of persistent objects currently loaded into TPM
            // NV memory
```



```
            *value = NvCapGetPersistentNumber();
            break;
        case TPM_PT_HR_PERSISTENT_AVAIL:
            // number of additional persistent objects that could be loaded
            // into NV memory
            *value = NvCapGetPersistentAvail();
            break;
        case TPM_PT_NV_COUNTERS:
            // number of defined NV indexes that have NV TPMA_NV_COUNTER
            // attribute SET
            *value = NvCapGetCounterNumber();
            break;
        case TPM_PT_NV_COUNTERS_AVAIL:
            // number of additional NV indexes that can be defined with their
            // TPMA_NV_COUNTER attribute SET
            *value = NvCapGetCounterAvail();
            break;
        case TPM_PT_ALGORITHM_SET:
            // region code for the TPM
            *value = gp.algorithmSet;
            break;

        case TPM_PT_LOADED_CURVES:
    #ifdef TPM_ALG_ECC
            // number of loaded ECC curves
            *value = CryptCapGetEccCurveNumber();
    #else // TPM_ALG_ECC
            *value = 0;
    #endif // TPM_ALG_ECC
            break;

        case TPM_PT_LOCKOUT_COUNTER:
            // current value of the lockout counter
            *value = gp.failedTries;
            break;
        case TPM_PT_MAX_AUTH_FAIL:
            // number of authorization failures before DA lockout is invoked
            *value = gp.maxTries;
            break;
        case TPM_PT_LOCKOUT_INTERVAL:
            // number of seconds before the value reported by
            // TPM_PT_LOCKOUT_COUNTER is decremented
            *value = gp.recoveryTime;
            break;
        case TPM_PT_LOCKOUT_RECOVERY:
            // number of seconds after a lockoutAuth failure before use of
            // lockoutAuth may be attempted again
            *value = gp.lockoutRecovery;
            break;
        case TPM_PT_AUDIT_COUNTER_0:
            // high-order 32 bits of the command audit counter
            *value = (UINT32) (gp.auditCounter >> 32);
            break;
        case TPM_PT_AUDIT_COUNTER_1:
            // low-order 32 bits of the command audit counter
            *value = (UINT32) (gp.auditCounter);
            break;
        default:
            // property is not defined
            return FALSE;
            break;
    }

    return TRUE;
}
```

10.14.3.2 TPMCapGetProperties()

This function is used to get the TPM_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Table 110

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```
TPMI_YES_NO
TPMCapGetProperties(
    TPM_PT                          property,       // IN: the starting TPM property
    UINT32                          count,          // IN: maximum number of returned
                                                    //     properties
    TPML_TAGGED_TPM_PROPERTY       *propertyList    // OUT: property list
    )
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    // initialize output property list
    propertyList->count = 0;

    // maximum count of properties we may return is MAX_PCR_PROPERTIES
    if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;

    // If property is less than PT_FIXED, start from PT_FIXED.
    if(property < PT_FIXED) property = PT_FIXED;

    // Scan through the TPM properties of the requested group.
    // The size of TPM property group is PT_GROUP * 2 for fix and
    // variable groups.
    for(i = property; i <= PT_FIXED + PT_GROUP * 2; i++)
    {
        UINT32          value;
        if(TPMPropertyIsDefined((TPM_PT) i, &value))
        {
            if(propertyList->count < count)
            {

                // If the list is not full, add this property
                propertyList->tpmProperty[propertyList->count].property =
                    (TPM_PT) i;
                propertyList->tpmProperty[propertyList->count].value = value;
                propertyList->count++;
            }
            else
            {
                // If the return list is full but there are more properties
                // available, set the indication and exit the loop.
                more = YES;
                break;
            }
        }
    }
    return more;
}
```


10.15 TpmFail.c

10.15.1 Includes, Defines, and Types

```
#define     TPM_FAIL_C
#include    "InternalRoutines.h"
#include    <assert.h>
```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TPM_Types.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TPM_Types.h and only include it for the #defines of the capabilities, properties, and command code values.

```
#pragma pack(push, 1)
#include "TPM_Types.h"
#pragma pack (pop)
#include "swap.h"
```

10.15.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```
#pragma pack(push,1)
typedef struct {
    TPM_ST          tag;
    UINT32          size;
    TPM_RC          code;
} HEADER;
typedef struct {
    UINT16      size;
    struct {
        UINT32      function;
        UINT32      line;
        UINT32      code;
    } values;
    TPM_RC      returnCode;
} GET_TEST_RESULT_PARAMETERS;
typedef struct {
    TPMI_YES_NO                 moreData;
    TPM_CAP                     capability; // Always TPM_CAP_TPM_PROPERTIES
    TPML_TAGGED_TPM_PROPERTY    tpmProperty; // a single tagged property
} GET_CAPABILITY_PARAMETERS;
typedef struct {
    HEADER header;
    GET_TEST_RESULT_PARAMETERS  getTestResult;
} TEST_RESPONSE;
typedef struct {
    HEADER header;
    GET_CAPABILITY_PARAMETERS   getCap;
} CAPABILITY_RESPONSE;
typedef union {
    TEST_RESPONSE           test;
    CAPABILITY_RESPONSE     cap;
} RESPONSES;
#pragma pack(pop)
```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode). There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```
#ifndef __IGNORE_STATE__ // Don't define this value
static BYTE response[sizeof(RESPONSES)];
#endif
```

10.15.3 Local Functions

10.15.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
static INT32
MarshalUint16(
    UINT16          integer,
    BYTE            **buffer
    )
{
    return UINT16_Marshal(&integer, buffer, NULL);
}
```

10.15.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
static INT32
MarshalUint32(
    UINT32           integer,
    BYTE            **buffer
    )
{
    return UINT32_Marshal(&integer, buffer, NULL);
}
```

10.15.3.3 UnmarshalHeader()

Funtion to unmarshal the 10-byte command header.

```
static BOOL
UnmarshalHeader(
    HEADER          *header,
    BYTE            **buffer,
    INT32           *size
    )
{
    UINT32 usize;
    TPM_RC ucode;
    if(     UINT16_Unmarshal(&header->tag, buffer, size) != TPM_RC_SUCCESS
        ||  UINT32_Unmarshal(&usize, buffer, size) != TPM_RC_SUCCESS
        ||  UINT32_Unmarshal(&ucode, buffer, size) != TPM_RC_SUCCESS
        )
        return FALSE;
    header->size = usize;
    header->code = ucode;
    return TRUE;
}
```

10.15.4 Public Functions

10.15.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```
LIB_EXPORT void
SetForceFailureMode(
    void
    )
{
    g_forceFailureMode = TRUE;
    return;
}
```

10.15.4.2 TpmFail()

This function is called by TPM. lib when a failure occurs. It will set up the failure values to be returned on TPM2_GetTestResult().

```
void
TpmFail(
    const char                    *function,
    int line,            int      code
    )
{
    // Save the values that indicate where the error occurred.
    // On a 64-bit machine, this may truncate the address of the string
    // of the function name where the error occurred.
    s_failFunction = *(UINT32*)&function;
    s_failLine = line;
    s_failCode = code;

    // if asserts are enabled, then do an assert unless the failure mode code
    // is being tested
    assert(g_forceFailureMode);

    // Clear this flag
    g_forceFailureMode = FALSE;

    // Jump to the failure mode code.
    // Note: only get here if asserts are off or if we are testing failure mode
    longjmp(&g_jumpBuffer[0], 1);
}
```


10.15.5 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```
void
TpmFailureMode (
    unsigned int     inRequestSize,      // IN: command buffer size
    unsigned char   *inRequest,          // IN: command buffer
    unsigned int    *outResponseSize,    // OUT: response buffer size
    unsigned char   **outResponse        // OUT: response buffer
    )
{
    BYTE            *buffer;
    UINT32           marshalSize;
    UINT32           capability;
    HEADER           header;    // unmarshaled command header
```

```
    UINT32          pt;    // unmarshaled property type
    UINT32          count; // unmarshaled property count

    // If there is no command buffer, then just return TPM_RC_FAILURE
    if(inRequestSize == 0 || inRequest == NULL)
        goto FailureModeReturn;

    // If the header is not correct for TPM2_GetCapability() or
    // TPM2_GetTestResult() then just return the in failure mode response;
    buffer = inRequest;
    if(!UnmarshalHeader(&header, &inRequest, (INT32 *)&inRequestSize))
        goto FailureModeReturn;
    if(   header.tag  != TPM_ST_NO_SESSIONS
       || header.size < 10)
       goto FailureModeReturn;

    switch (header.code) {
    case TPM_CC_GetTestResult:

        // make sure that the command size is correct
        if(header.size != 10)
            goto FailureModeReturn;
        buffer = &response[10];
        marshalSize =  MarshalUint16(3 * sizeof(UINT32), &buffer);
        marshalSize += MarshalUint32(s_failFunction, &buffer);
        marshalSize += MarshalUint32(s_failLine, &buffer);
        marshalSize += MarshalUint32(s_failCode, &buffer);
        if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
            marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
        else
            marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
        break;

    case TPM_CC_GetCapability:
        // make sure that the size of the command is exactly the size
        // returned for the capability, property, and count
        if(     header.size!= (10 + (3 * sizeof(UINT32)))
                // also verify that this is requesting TPM properties
            ||      (UINT32_Unmarshal(&capability, &inRequest,
                                      (INT32 *)&inRequestSize)
                != TPM_RC_SUCCESS)
            || (capability != TPM_CAP_TPM_PROPERTIES)
            ||      (UINT32_Unmarshal(&pt, &inRequest, (INT32 *)&inRequestSize)
                != TPM_RC_SUCCESS)
            ||      (UINT32_Unmarshal(&count, &inRequest, (INT32 *)&inRequestSize)
                != TPM_RC_SUCCESS)
            )

            goto FailureModeReturn;


        // If in failure mode because of an unrecoverable read error, and the
        // property is 0 and the count is 0, then this is an indication to
        // re-manufacture the TPM. Do the re-manufacture but stay in failure
        // mode until the TPM is reset.
        // Note: this behavior is not required by ISO/IEC 11889 and it is
        // OK to leave the TPM permanently bricked due to an unrecoverable NV
        // error.
        if( count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
        {
            g_manufactured = FALSE;
            TPM_Manufacture(0);
        }

        if(count > 0)
            count = 1;
```



```
        else if(pt > TPM_PT_FIRMWARE_VERSION_2)
            count = 0;
        if(pt < TPM_PT_MANUFACTURER)
            pt = TPM_PT_MANUFACTURER;

        // set up for return
        buffer = &response[10];
        // if the request was for a PT less than the last one
        // then we indicate more, otherwise, not.
        if(pt < TPM_PT_FIRMWARE_VERSION_2)
            *buffer++ = YES;
        else
            *buffer++ = NO;

        marshalSize = 1;

        // indicate the capability type
        marshalSize += MarshalUint32(capability, &buffer);
        // indicate the number of values that are being returned (0 or 1)
        marshalSize += MarshalUint32(count, &buffer);
        // indicate the property
        marshalSize += MarshalUint32(pt, &buffer);

        if(count > 0)
            switch (pt) {
            case TPM_PT_MANUFACTURER:
            // the vendor ID unique to each TPM manufacturer
#ifdef  MANUFACTURER
            pt = *(UINT32*)MANUFACTURER;
#else
            pt = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_1:
            // the first four characters of the vendor ID string
#ifdef  VENDOR_STRING_1
            pt = *(UINT32*)VENDOR_STRING_1;
#else
            pt = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_2:
            // the second four characters of the vendor ID string
#ifdef  VENDOR_STRING_2
            pt = *(UINT32*)VENDOR_STRING_2;
#else
            pt = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_3:
            // the third four characters of the vendor ID string
#ifdef  VENDOR_STRING_3
            pt = *(UINT32*)VENDOR_STRING_3;
#else
            pt = 0;
#endif
            break;
        case TPM_PT_VENDOR_STRING_4:
            // the fourth four characters of the vendor ID string
#ifdef  VENDOR_STRING_4
            pt = *(UINT32*)VENDOR_STRING_4;
#else
            pt = 0;
#endif

            break;
```

```
        case TPM_PT_VENDOR_TPM_TYPE:
            // vendor-defined value indicating the TPM model
            // We just make up a number here
            pt = 1;
            break;
        case TPM_PT_FIRMWARE_VERSION_1:
            // the more significant 32-bits of a vendor-specific value
            // indicating the version of the firmware
#ifdef  FIRMWARE_V1
            pt = FIRMWARE_V1;
#else
            pt = 0;
#endif
            break;
        default: // TPM_PT_FIRMWARE_VERSION_2:
            // the less significant 32-bits of a vendor-specific value
            // indicating the version of the firmware
#ifdef  FIRMWARE_V2
            pt = FIRMWARE_V2;
#else
            pt = 0;
#endif
            break;
        }
        marshalSize += MarshalUint32(pt, &buffer);
        break;
    default: // default for switch (cc)
        goto FailureModeReturn;
    }
    // Now do the header
    buffer = response;
    marshalSize = marshalSize + 10; // Add the header size to the
                                    // stuff already marshaled
    MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
    MarshalUint32(marshalSize, &buffer); // responseSize
    MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code

    *outResponseSize = marshalSize;
    *outResponse = (unsigned char *)&response;
    return;

FailureModeReturn:

    buffer = response;

    marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
    marshalSize += MarshalUint32(10, &buffer);
    marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);

    *outResponseSize = marshalSize;
    *outResponse = (unsigned char *)response;
    return;
}
```

11 Cryptographic Functions

11.1 Introduction

The files in clause 11 provide cryptographic support for the other functions in the TPM and the interface to the Crypto Engine.

Per the ISO/IEC 11889-1, clause 11.5, ”Authorization Subsystem” support for HMAC is mandatory and HMAC is defined in ISO/IEC 9797-2, making ISO/IEC 9797-2 indispensable for implementation of the required cryptographic functions for this International Standard.

Per the ISO/IEC 11889-1, clause 11.4.6.1, “Introduction” support for Cipher Feedback mode (CFB) is mandatory. CFB is defined ISO/IEC 10116:2006, making ISO/IEC 10116:2006 indispensable for implementation of the required cryptographic functions for this International Standard.

11.2 CryptUtil.c

11.2.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

11.2.2 Includes

```
#include     "TPM_Types.h"
#include     "CryptoEngine.h"    // types shared by CryptUtil and CryptoEngine.
                                 // Includes the function prototypes for the
                                 // CryptoEngine functions.
#include     "Global.h"
#include     "InternalRoutines.h"
#include     "MemoryLib_fp.h"
//#include     "CryptSelfTest_fp.h"
```

11.2.3 TranslateCryptErrors()

This function converts errors from the cryptographic library into TPM_RC_VALUES.

Table 111

Error Returns	Meaning
TPM_RC_VALUE	CRYPT_FAIL
TPM_RC_NO_RESULT	CRYPT_NO_RESULT
TPM_RC_SCHEME	CRYPT_SCHEME
TPM_RC_VALUE	CRYPT_PARAMETER
TPM_RC_SIZE	CRYPT_UNDERFLOW
TPM_RC_ECC_POINT	CRYPT_POINT
TPM_RC_CANCELLED	CRYPT_CANCEL

```
static TPM_RC
TranslateCryptErrors (
    CRYPT_RESULT          retVal             // IN: crypt error to evaluate
)
```

```
{
    switch (retVal)
    {
    case CRYPT_SUCCESS:
        return TPM_RC_SUCCESS;
    case CRYPT_FAIL:
        return TPM_RC_VALUE;
    case CRYPT_NO_RESULT:
        return TPM_RC_NO_RESULT;
    case CRYPT_SCHEME:
        return TPM_RC_SCHEME;
    case CRYPT_PARAMETER:
        return TPM_RC_VALUE;
    case CRYPT_UNDERFLOW:
        return TPM_RC_SIZE;
    case CRYPT_POINT:
        return TPM_RC_ECC_POINT;
    case CRYPT_CANCEL:
        return TPM_RC_CANCELED;
    default: // Other unknown warnings
        return TPM_RC_FAILURE;
    }
}
```

11.2.4 Random Number Generation Functions

11.2.4.1 Preamble

```
#ifdef TPM_ALG_NULL //%
#ifdef _DRBG_STATE_SAVE //%
```

11.2.4.2 CryptDrbgGetPutState()

Read or write the current state from the DRBG in the *cryptoEngine*.

```
void
CryptDrbgGetPutState(
    GET_PUT          direction      // IN: Get from or put to DRBG
    )
{
    _cpri__DrbgGetPutState(direction,
                           sizeof(go.drbgState),
                           (BYTE *)&go.drbgState);
}
#else   //% 00
//%#define CryptDrbgGetPutState(ignored)  // If not doing state save, turn this
//%                                        // into a null macro
#endif  //%
```

11.2.4.3 CryptStirRandom()

Stir random entropy

```
void
CryptStirRandom(
    UINT32           entropySize,   // IN: size of entropy buffer
    BYTE            *buffer         // IN: entropy buffer
    )
{
    // RNG self testing code may be inserted here

```

```
    // Call crypto engine random number stirring function
    _cpri__StirRandom(entropySize, buffer);

    return;
}
```

11.2.4.4 CryptGenerateRandom()

This is the interface to _cpri__GenerateRandom().

```
UINT16
CryptGenerateRandom(
    UINT16           randomSize,     // IN: size of random number
    BYTE            *buffer          // OUT: buffer of random number
    )
{
    UINT16               result;
    pAssert(randomSize <= MAX_RSA_KEY_BYTES || randomSize <= PRIMARY_SEED_SIZE);
    if(randomSize == 0)
        return 0;

    // Call crypto engine random number generation
    result = _cpri__GenerateRandom(randomSize, buffer);
    if(result != randomSize)
        FAIL(FATAL_ERROR_INTERNAL);

    return result;
}
#endif //TPM_ALG_NULL //%
```

11.2.5 Hash/HMAC Functions

11.2.5.1 CryptGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```
#ifdef TPM_ALG_KEYEDHASH        //% 1
TPM_ALG_ID
CryptGetContextAlg(
    void            *state          // IN: the context to check
    )
{
    HASH_STATE   *context = (HASH_STATE *)state;
    return _cpri__GetContextAlg(&context->state);
}
```

11.2.5.2 CryptStartHash()

This function starts a hash and return the size, in bytes, of the digest.

Table 112

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
UINT16
CryptStartHash(
```

```
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    HASH_STATE      *hashState       // OUT: the state of hash stack. It will be used
                                     //     in hash update and completion
    )
{
    CRYPT_RESULT       retVal = 0;

    pAssert(hashState != NULL);

    TEST_HASH(hashAlg);

    hashState->type = HASH_STATE_EMPTY;

    // Call crypto engine start hash function
    if((retVal = _cpri__StartHash(hashAlg, FALSE, &hashState->state)) > 0)
        hashState->type = HASH_STATE_HASH;

    return retVal;
}
```

11.2.5.3 CryptStartHashSequence()

Start a hash stack for a sequence object and return the size, in bytes, of the digest. This call uses the form of the hash state that requires context save and restored.

Table 113

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
UINT16
CryptStartHashSequence(
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    HASH_STATE      *hashState       // OUT: the state of hash stack. It will be used
                                     //     in hash update and completion
    )
{
    CRYPT_RESULT    retVal = 0;

    pAssert(hashState != NULL);

    TEST_HASH(hashAlg);

    hashState->type = HASH_STATE_EMPTY;

    // Call crypto engine start hash function
    if((retVal = _cpri__StartHash(hashAlg, TRUE, &hashState->state)) > 0)
        hashState->type = HASH_STATE_HASH;

    return retVal;

}
```

11.2.5.4 CryptStartHMAC()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Table 114

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
UINT16
CryptStartHMAC(
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    UINT16           keySize,        // IN: the size of HMAC key in bytes
    BYTE            *key,            // IN: HMAC key
    HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
                                     //     in HMAC update and completion
    )
{
    HASH_STATE      *hashState = (HASH_STATE *)hmacState;
    CRYPT_RESULT    retVal;

    // This has to come before the pAssert in case we all calling this function
    // during testing. If so, the first instance will have no arguments but the
    // hash algorithm. The call from the test routine will have arguments. When
    // the second call is done, then we return to the test dispatcher.
    TEST_HASH(hashAlg);

    pAssert(hashState != NULL);

    hashState->type = HASH_STATE_EMPTY;

    if((retVal =  _cpri__StartHMAC(hashAlg, FALSE, &hashState->state, keySize, key,
                                &hmacState->hmacKey.b)) > 0)
        hashState->type = HASH_STATE_HMAC;

    return retVal;
}
```

11.2.5.5 CryptStartHMACSequence()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

This call is used to start a sequence HMAC that spans multiple TPM commands.

Table 115

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
UINT16
CryptStartHMACSequence(
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    UINT16           keySize,        // IN: the size of HMAC key in bytes
    BYTE            *key,            // IN: HMAC key
    HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
                                     //     in HMAC update and completion
    )
{
    HASH_STATE      *hashState = (HASH_STATE *)hmacState;
```

```
    CRYPT_RESULT    retVal;

    TEST_HASH(hashAlg);

    hashState->type = HASH_STATE_EMPTY;

    if((retVal =  _cpri__StartHMAC(hashAlg, TRUE, &hashState->state,
                               keySize, key, &hmacState->hmacKey.b)) > 0)
        hashState->type = HASH_STATE_HMAC;

    return retVal;
}
```

11.2.5.6 CryptStartHMAC2B()

This function starts an HMAC and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Table 116

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
LIB_EXPORT UINT16
CryptStartHMAC2B(
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    TPM2B           *key,            // IN: HMAC key
    HMAC_STATE      *hmacState       // OUT: the state of HMAC stack. It will be used
                                     //     in HMAC update and completion
    )
{
    return CryptStartHMAC(hashAlg, key->size, key->buffer, hmacState);
}
```

11.2.5.7 CryptStartHMACSequence2B()

This function starts an HMAC sequence and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Table 117

Return Value	Meaning
> 0	the digest size of the algorithm
= 0	the *hashAlg* was TPM_ALG_NULL

```
UINT16
CryptStartHMACSequence2B(
    TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
    TPM2B           *key,            // IN: HMAC key
```



```
    HMAC_STATE      *hmacState      // OUT: the state of HMAC stack. It will be used
                                    //     in HMAC update and completion
    )
{
    return CryptStartHMACSequence(hashAlg, key->size, key->buffer, hmacState);
}
```

11.2.5.8 CryptUpdateDigest()

This function updates a digest (hash or HMAC) with an array of octets.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
LIB_EXPORT void
CryptUpdateDigest(
    void            *digestState,   // IN: the state of hash stack
    UINT32           dataSize,      // IN: the size of data
    BYTE            *data           // IN: data to be hashed
    )
{
    HASH_STATE      *hashState = (HASH_STATE *)digestState;

    pAssert(digestState != NULL);

    if(hashState->type != HASH_STATE_EMPTY && data != NULL && dataSize != 0)
    {
        // Call crypto engine update hash function
        _cpri__UpdateHash(&hashState->state, dataSize, data);
    }
    return;
}
```

11.2.5.9 CryptUpdateDigest2B()

This function updates a digest (hash or HMAC) with a TPM2B.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```
LIB_EXPORT void
CryptUpdateDigest2B(
    void            *digestState,   // IN: the digest state
    TPM2B           *bIn            // IN: 2B containing the data
    )
{
    // Only compute the digest if a pointer to the 2B is provided.
    // In CryptUpdateDigest(), if size is zero or buffer is NULL, then no change
    // to the digest occurs. This function should not provide a buffer if bIn is
    // not provided.
    if(bIn != NULL)
        CryptUpdateDigest(digestState, bIn->size, bIn->buffer);
    return;
}
```

11.2.5.10 CryptUpdateDigestInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptUpdateHash().

```
LIB_EXPORT void
```

```
CryptUpdateDigestInt(
    void              *state,         // IN: the state of hash stack
    UINT32             intSize,       // IN: the size of 'intValue' in bytes
    void              *intValue       // IN: integer value to be hashed
    )
{

#if BIG_ENDIAN_TPM == YES
    pAssert(    intValue != NULL && (intSize == 1 || intSize == 2
            ||  intSize == 4 || intSize == 8));
    CryptUpdateHash(state, inSize, (BYTE *)intValue);
#else

    BYTE        marshalBuffer[8];
    // Point to the big end of an little-endian value
    BYTE        *p = &((BYTE *)intValue)[intSize - 1];
    // Point to the big end of an big-endian value
    BYTE        *q = marshalBuffer;

    pAssert(intValue != NULL);
    switch (intSize)
    {
    case 8:
        *q++ = *p--;
        *q++ = *p--;
        *q++ = *p--;
        *q++ = *p--;
    case 4:
        *q++ = *p--;
        *q++ = *p--;
    case 2:
        *q++ = *p--;
    case 1:
        *q = *p;
        // Call update the hash
        CryptUpdateDigest(state, intSize, marshalBuffer);
        break;
    default:
        FAIL(0);
    }

#endif
    return;
}
```

11.2.5.11 CryptCompleteHash()

This function completes a hash sequence and returns the digest.

This function can be called to complete either an HMAC or hash sequence. The state type determines if the context type is a hash or HMAC. If an HMAC, then the call is forwarded to CryptCompleteHash().

If **digestSize** is smaller than the digest size of hash/HMAC algorithm, the most significant bytes of required size will be returned.

Table 118

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
LIB_EXPORT UINT16
CryptCompleteHash(
    void              *state,         // IN: the state of hash stack
```



```
    UINT16          digestSize,     // IN: size of digest buffer
    BYTE           *digest          // OUT: hash digest
    )
{
    HASH_STATE     *hashState = (HASH_STATE *)state;     // local value

    // If the session type is HMAC, then could forward this to
    // the HMAC processing and not cause an error. However, if no
    // function calls this routine to forward it, then we can't get
    // test coverage. The decision is to assert if this is called with
    // the type == HMAC and fix anything that makes the wrong call.
    pAssert(hashState->type == HASH_STATE_HASH);

    // Set the state to empty so that it doesn't get used again
    hashState->type = HASH_STATE_EMPTY;

    // Call crypto engine complete hash function
    return     _cpri__CompleteHash(&hashState->state, digestSize, digest);
}
```

11.2.5.12 CryptCompleteHash2B()

This function is the same as CypteCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for clarity for this part of ISO/IEC 11889. 'digest.size' should be set to indicate the number of bytes to place in the buffer.

Table 119

Return Value	Meaning
>=0	the number of bytes placed in 'digest.buffer'

```
LIB_EXPORT UINT16
CryptCompleteHash2B(
    void           *state,          // IN: the state of hash stack
    TPM2B          *digest          // IN: the size of the buffer Out: requested
                                    //     number of bytes

    )
{
    UINT16              retVal = 0;

    if(digest != NULL)
        retVal = CryptCompleteHash(state, digest->size, digest->buffer);

    return retVal;
}
```

11.2.5.13 CryptHashBlock()

Hash a block of data and return the results. If the digest is larger than *retSize*, it is truncated and with the least significant octets dropped.

Table 120

Return Value	Meaning
>=0	the number of bytes placed in *ret*

```
LIB_EXPORT UINT16
CryptHashBlock(
    TPM_ALG_ID      algId,          // IN: the hash algorithm to use
```

```
    UINT16           blockSize,     // IN: size of the data block
    BYTE            *block,         // IN: address of the block to hash
    UINT16           retSize,       // IN: size of the return buffer
    BYTE            *ret            // OUT: address of the buffer
    )
{
    TEST_HASH(algId);

    return _cpri__HashBlock(algId, blockSize, block, retSize, ret);
}
```

11.2.5.14 CryptCompleteHMAC()

This function completes a HMAC sequence and returns the digest. If *digestSize* is smaller than the digest size of the HMAC algorithm, the most significant bytes of required size will be returned.

Table 121

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
LIB_EXPORT UINT16
CryptCompleteHMAC(
    HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
    UINT32           digestSize,    // IN: size of digest buffer
    BYTE            *digest         // OUT: HMAC digest
    )
{
    HASH_STATE      *hashState;

    pAssert(hmacState != NULL);
    hashState = &hmacState->hashState;

    pAssert(hashState->type == HASH_STATE_HMAC);

    hashState->type = HASH_STATE_EMPTY;

    return _cpri__CompleteHMAC(&hashState->state, &hmacState->hmacKey.b,
                               digestSize, digest);

}
```


11.2.5.15 CryptCompleteHMAC2B()

This function is the same as CryptCompleteHMAC() but the HMAC result is returned in a TPM2B which is the most common use.

Table 122

Return Value	Meaning
>=0	the number of bytes placed in *digest*

```
LIB_EXPORT UINT16
CryptCompleteHMAC2B(
    HMAC_STATE      *hmacState,     // IN: the state of HMAC stack
    TPM2B           *digest         // OUT: HMAC
    )
{
    UINT16               retVal = 0;
```



```
    if(digest != NULL)
        retVal = CryptCompleteHMAC(hmacState, digest->size, digest->buffer);
    return retVal;
}
```

11.2.5.16 CryptHashStateImportExport()

This function is used to prepare a hash state context for LIB_EXPORT or to import it into the internal format. It is used by TPM2_ContextSave() and TPM2_ContextLoad() via SequenceDataImportExport(). This is just a pass-through function to the crypto library.

```
void
CryptHashStateImportExport(
    HASH_STATE      *internalFmt,    // IN: state to LIB_EXPORT
    HASH_STATE      *externalFmt,    // OUT: exported state
    IMPORT_EXPORT    direction
    )
{
    _cpri__ImportExportHashState(&internalFmt->state,
                                 (EXPORT_HASH_STATE *)&externalFmt->state,
                                 direction);
}
```

11.2.5.17 CryptGetHashDigestSize()

This function returns the digest size in bytes for a hash algorithm.

Table 123

Return Value	Meaning
0	digest size for TPM_ALG_NULL
> 0	digest size

```
LIB_EXPORT UINT16
CryptGetHashDigestSize(
    TPM_ALG_ID       hashAlg         // IN: hash algorithm
    )
{
    return _cpri__GetDigestSize(hashAlg);
}
```

11.2.5.18 CryptGetHashBlockSize()

Get the digest size in byte of a hash algorithm.

Table 124

Return Value	Meaning
0	block size for TPM_ALG_NULL
> 0	block size

```
LIB_EXPORT UINT16
CryptGetHashBlockSize(
    TPM_ALG_ID       hash            // IN: hash algorithm to look up
    )
{
```

```
    return _cpri__GetHashBlockSize(hash);
}
```

11.2.5.19 CryptGetHashAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* value of 2 will return the last implemented hash. All other index values will return TPM_ALG_NULL.

Table 125

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
LIB_EXPORT TPM_ALG_ID
CryptGetHashAlgByIndex(
    UINT32              index           // IN: the index
    )
{
    return _cpri__GetHashAlgByIndex(index);
}
```

11.2.5.20 CryptSignHMAC()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Table 126

Error Returns	Meaning

```
static TPM_RC
CryptSignHMAC(
    OBJECT              *signKey,       // IN: HMAC key sign the hash
    TPMT_SIG_SCHEME     *scheme,        // IN: signing scheme
    TPM2B_DIGEST        *hashData,      // IN: hash to be signed
    TPMT_SIGNATURE      *signature      // OUT: signature
    )
{
    HMAC_STATE          hmacState;
    UINT32              digestSize;

    // HMAC algorithm self testing code may be inserted here

    digestSize = CryptStartHMAC2B(scheme->details.hmac.hashAlg,
                                  &signKey->sensitive.sensitive.bits.b,
                                  &hmacState);

    // The hash algorithm must be a valid one.
    pAssert(digestSize > 0);

    CryptUpdateDigest2B(&hmacState, &hashData->b);

    CryptCompleteHMAC(&hmacState, digestSize,
                      (BYTE *) &signature->signature.hmac.digest);

    // Set HMAC algorithm
    signature->signature.hmac.hashAlg = scheme->details.hmac.hashAlg;
```



```
    return TPM_RC_SUCCESS;
}
```

11.2.5.21 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key.

Table 127

Error Returns	Meaning
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
static TPM_RC
CryptHMACVerifySignature(
    OBJECT          *signKey,       // IN: HMAC key signed the hash
    TPM2B_DIGEST    *hashData,      // IN: digest being verified
    TPMT_SIGNATURE  *signature      // IN: signature to be verified
    )
{
    HMAC_STATE          hmacState;
    TPM2B_DIGEST        digestToCompare;

    digestToCompare.t.size = CryptStartHMAC2B(signature->signature.hmac.hashAlg,
                             &signKey->sensitive.sensitive.bits.b, &hmacState);

    CryptUpdateDigest2B(&hmacState, &hashData->b);

    CryptCompleteHMAC2B(&hmacState, &digestToCompare.b);

    // Compare digest
    if(MemoryEqual(digestToCompare.t.buffer,
                   (BYTE *) &signature->signature.hmac.digest,
                   digestToCompare.t.size))
        return TPM_RC_SUCCESS;
    else
        return TPM_RC_SIGNATURE;

}
```

11.2.5.22 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Table 128

Error Returns	Meaning
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```
static TPM_RC
CryptGenerateKeyedHash(
    TPMT_PUBLIC               *publicArea,        // IN/OUT: the public area template
                                                  //     for the new key.
    TPMS_SENSITIVE_CREATE     *sensitiveCreate,   // IN: sensitive creation data
    TPMT_SENSITIVE            *sensitive,         // OUT: sensitive area
    TPM_ALG_ID                 kdfHashAlg,        // IN: algorithm for the KDF
    TPM2B_SEED                *seed,              // IN: the seed
    TPM2B_NAME                *name               // IN: name of the object
    )
```

```
{
    TPMT_KEYEDHASH_SCHEME   *scheme;
    TPM_ALG_ID               hashAlg;
    UINT16                   hashBlockSize;

    scheme = &publicArea->parameters.keyedHashDetail.scheme;

    pAssert(publicArea->type == TPM_ALG_KEYEDHASH);

    // Pick the limiting hash algorithm
    if(scheme->scheme == TPM_ALG_NULL)
        hashAlg = publicArea->nameAlg;
    else if(scheme->scheme == TPM_ALG_XOR)
        hashAlg = scheme->details.xor.hashAlg;
    else
        hashAlg = scheme->details.hmac.hashAlg;
    hashBlockSize =  CryptGetHashBlockSize(hashAlg);

    // if this is a signing or a decryption key, then then the limit
    // for the data size is the block size of the hash. This limit
    // is set because larger values have lower entropy because of the
    // HMAC function.
    if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
    {
        if(    (   publicArea->objectAttributes.decrypt
                || publicArea->objectAttributes.sign)
            && sensitiveCreate->data.t.size > hashBlockSize)

            return TPM_RC_SIZE;
    }
    else
    {
        // If the TPM is going to generate the data, then set the size to be the
        // size of the digest of the algorithm
        sensitive->sensitive.sym.t.size = CryptGetHashDigestSize(hashAlg);
        sensitiveCreate->data.t.size = 0;
    }

    // Fill in the sensitive area
    CryptGenerateNewSymmetric(sensitiveCreate, sensitive, kdfHashAlg,
                              seed, name);

    // Create unique area in public
    CryptComputeSymmetricUnique(publicArea->nameAlg,
                                sensitive, &publicArea->unique.sym);

    return TPM_RC_SUCCESS;
}
```

11.2.5.23 CryptKDFa()

This function generates a key using the KDFa() formulation in ISO/IEC 11889-1. In this implementation, this is a macro invocation of _cpri__KDFa() in the hash module of the CryptoEngine(). This macro sets *once* to FALSE so that KDFa() will iterate as many times as necessary to generate *sizeInBits* number of bits.

```
//%#define CryptKDFa(hashAlg, key, label, contextU, contextV,    \
//%                  sizeInBits, keyStream, counterInOut)         \
//%         TEST_HASH(hashAlg);                                   \
//%        _cpri__KDFa(                                           \
//%                        ((TPM_ALG_ID)hashAlg),                 \
//%                        ((TPM2B *)key),                        \
//%                        ((const char *)label),                 \
//%                        ((TPM2B *)contextU),                   \
```

```
//%                    ((TPM2B *)contextV),                   \
//%                    ((UINT32)sizeInBits),                  \
//%                    ((BYTE *)keyStream),                   \
//%                    ((UINT32 *)counterInOut),              \
//%                    ((BOOL) FALSE)                         \
//%                   )
//%
```

11.2.5.24 CryptKDFaOnce()

This function generates a key using the KDFa() formulation in ISO/IEC 11889-1. In this implementation, this is a macro invocation of _cpri__KDFa() in the hash module of the CryptoEngine(). This macro will call _cpri__KDFa() with **once** TRUE so that only one iteration is performed, regardless of *sizeInBits*.

```
//%#define CryptKDFaOnce(hashAlg, key, label, contextU, contextV,    \
//%                      sizeInBits, keyStream, counterInOut)       \
//%         TEST_HASH(hashAlg);                                     \
//%        _cpri__KDFa(                                             \
//%                    ((TPM_ALG_ID)hashAlg),                       \
//%                    ((TPM2B *)key),                              \
//%                    ((const char *)label),                       \
//%                    ((TPM2B *)contextU),                         \
//%                    ((TPM2B *)contextV),                         \
//%                    ((UINT32)sizeInBits),                        \
//%                    ((BYTE *)keyStream),                         \
//%                    ((UINT32 *)counterInOut),                    \
//%                    ((BOOL) TRUE)                                \
//%                   )
//%
```

11.2.5.25 KDFa()

This function is used by functions outside of CryptUtil() to access _cpri_KDFa().

```
void
KDFa(
    TPM_ALG_ID       hash,          // IN: hash algorithm used in HMAC
    TPM2B           *key,           // IN: HMAC key
    const char      *label,         // IN: a null-terminated label for KDF
    TPM2B           *contextU,      // IN: context U
    TPM2B           *contextV,      // IN: context V
    UINT32           sizeInBits,    // IN: size of generated key in bits
    BYTE            *keyStream,     // OUT: key buffer
    UINT32          *counterInOut   // IN/OUT: caller may provide the iteration
                                    //     counter for incremental operations to
                                    //     avoid large intermediate buffers.
    )
{
    CryptKDFa(hash, key, label, contextU, contextV, sizeInBits,
              keyStream, counterInOut);
}
```

11.2.5.26 CryptKDFe()

This function generates a key using the KDFa() formulation in ISO/IEC 11889. In this implementation, this is a macro invocation of _cpri__KDFe() in the hash module of the CryptoEngine().

```
//%#define CryptKDFe(hashAlg, Z, label, partyUInfo, partyVInfo,          \
//%                  sizeInBits, keyStream)                             \
//% TEST_HASH(hashAlg);                                                 \
//% _cpri__KDFe(                                                        \
```



```
//%            ((TPM_ALG_ID)hashAlg),                                   \
//%            ((TPM2B *)Z),                                            \
//%            ((const char *)label),                                   \
//%            ((TPM2B *)partyUInfo),                                   \
//%            ((TPM2B *)partyVInfo),                                   \
//%            ((UINT32)sizeInBits),                                    \
//%            ((BYTE *)keyStream)                                      \
//%            )
//%
#endif //TPM_ALG_KEYEDHASH    //% 1
```

11.2.6 RSA Functions

11.2.6.1 BuildRSA()

Function to set the cryptographic elements of an RSA key into a structure to simplify the interface to _cpri__ RSA function. This can/should be eliminated by building this structure into the object structure.

```
#ifdef TPM_ALG_RSA          //% 2
static void
BuildRSA(
    OBJECT          *rsaKey,
    RSA_KEY         *key
    )
{
    key->exponent = rsaKey->publicArea.parameters.rsaDetail.exponent;
    if(key->exponent == 0)
        key->exponent = RSA_DEFAULT_PUBLIC_EXPONENT;
    key->publicKey = &rsaKey->publicArea.unique.rsa.b;

    if(rsaKey->attributes.publicOnly || rsaKey->privateExponent.t.size == 0)
        key->privateKey = NULL;
    else
        key->privateKey = &(rsaKey->privateExponent.b);
}
```

11.2.6.2 CryptTestKeyRSA()

This function provides the interface to _cpri__TestKeyRSA(). If both *p* and *q* are provided, *n* will be set to *p***q*.

If only *p* is provided, *q* is computed by *q* = *n*/*p*. If *n* mod *p* != 0, TPM_RC_BINDING is returned.

The key is validated by checking that a *d* can be found such that *e d* mod ((*p*-1)*(*q*-1)) = 1. If *d* is found that satisfies this requirement, it will be placed in *d*.

Table 129

Error Returns	Meaning
TPM_RC_BINDING	the public and private portions of the key are not matched

```
TPM_RC
CryptTestKeyRSA(
    TPM2B           *d,             // OUT: receives the private exponent
    UINT32           e,             // IN: public exponent
    TPM2B           *n,             // IN/OUT: public modulus
    TPM2B           *p,             // IN: a first prime
    TPM2B           *q              // IN: an optional second prime
    )
{
```



```
    CRYPT_RESULT    retVal;

    TEST(ALG_NULL_VALUE);

    pAssert(d != NULL && n != NULL && p != NULL);
    // Set the exponent
    if(e == 0)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;
    // CRYPT_PARAMETER
    retVal =_cpri__TestKeyRSA(d, e, n, p, q);
    if(retVal == CRYPT_SUCCESS)
        return TPM_RC_SUCCESS;
    else
        return TPM_RC_BINDING;  // convert CRYPT_PARAMETER
}
```

11.2.6.3 CryptGenerateKeyRSA()

This function is called to generate an RSA key from a provided seed. It calls _cpri__GenerateKeyRSA() to perform the computations. The implementation is vendor specific.

Table 130

Error Returns	Meaning
TPM_RC_RANGE	the exponent value is not supported
TPM_RC_CANCELLED	key generation has been canceled
TPM_RC_VALUE	exponent is not prime or is less than 3; or could not find a prime using the provided parameters

```
static TPM_RC
CryptGenerateKeyRSA(
    TPMT_PUBLIC         *publicArea,    // IN/OUT: The public area template for
                                        //      the new key. The public key
                                        //      area will be replaced by the
                                        //      product of two primes found by
                                        //      this function
    TPMT_SENSITIVE      *sensitive,     // OUT: the sensitive area will be
                                        //      updated to contain the first
                                        //      prime and the symmetric
                                        //      encryption key
    TPM_ALG_ID           hashAlg,       // IN: the hash algorithm for the KDF
    TPM2B_SEED          *seed,          // IN: Seed for the creation
    TPM2B_NAME          *name,          // IN: Object name
    UINT32              *counter        // OUT: last iteration of the counter
)
{
    CRYPT_RESULT    retVal;
    UINT32          exponent = publicArea->parameters.rsaDetail.exponent;

    TEST_HASH(hashAlg);
    TEST(ALG_NULL_VALUE);

    // In this implementation, only the default exponent is allowed
    if(exponent != 0 && exponent != RSA_DEFAULT_PUBLIC_EXPONENT)
        return TPM_RC_RANGE;
    exponent = RSA_DEFAULT_PUBLIC_EXPONENT;

    *counter = 0;

    // _cpri_GenerateKeyRSA can return CRYPT_CANCEL or CRYPT_FAIL
    retVal = _cpri__GenerateKeyRSA(&publicArea->unique.rsa.b,
```

```
                                        &sensitive->sensitive.rsa.b,
                                        publicArea->parameters.rsaDetail.keyBits,
                                        exponent,
                                        hashAlg,
                                        &seed->b,
                                        "RSA key by vendor",
                                        &name->b,
                                        counter);

    // CRYPT_CANCEL -> TPM_RC_CANCELLED; CRYPT_FAIL -> TPM_RC_VALUE
    return TranslateCryptErrors(retVal);

}
```

11.2.6.4 CryptLoadPrivateRSA()

This function is called to generate the private exponent of an RSA key. It uses CryptTestKeyRSA().

Table 131

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of *rsaKey* are not matched

```
TPM_RC
CryptLoadPrivateRSA(
    OBJECT          *rsaKey         // IN: the RSA key object
    )
{
    TPM_RC           result;
    TPMT_PUBLIC     *publicArea = &rsaKey->publicArea;
    TPMT_SENSITIVE  *sensitive = &rsaKey->sensitive;

    // Load key by computing the private exponent
    // TPM_RC_BINDING
    result = CryptTestKeyRSA(&(rsaKey->privateExponent.b),
                             publicArea->parameters.rsaDetail.exponent,
                             &(publicArea->unique.rsa.b),
                             &(sensitive->sensitive.rsa.b),
                             NULL);
    if(result == TPM_RC_SUCCESS)
        rsaKey->attributes.privateExp = SET;

    return result;
}
```

11.2.6.5 CryptSelectRSAScheme()

This function is used by TPM2_RSA_Decrypt() and TPM2_RSA_Encrypt(). It sets up the rules to select a scheme between input and object default. This function assume the RSA object is loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both the object and *scheme* are not TPM_ALG_NULL, then if the schemes are the same, the input scheme will be chosen. if the scheme are not compatible, a NULL pointer will be returned.

The return pointer may point to a TPM_ALG_NULL scheme.

```
TPMT_RSA_DECRYPT*
CryptSelectRSAScheme(
    TPMI_DH_OBJECT      rsaHandle,      // IN: handle of sign key
    TPMT_RSA_DECRYPT   *scheme          // IN: a sign or decrypt scheme
```



```
    )
{
    OBJECT              *rsaObject;
    TPMT_ASYM_SCHEME    *keyScheme;
    TPMT_RSA_DECRYPT    *retVal = NULL;

    // Get sign object pointer
    rsaObject = ObjectGet(rsaHandle);
    keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;

    // if the default scheme of the object is TPM_ALG_NULL, then select the
    // input scheme
    if(keyScheme->scheme == TPM_ALG_NULL)
    {
        retVal = scheme;
    }
    // if the object scheme is not TPM_ALG_NULL and the input scheme is
    // TPM_ALG_NULL, then select the default scheme of the object.
    else if(scheme->scheme == TPM_ALG_NULL)
    {
        // if input scheme is NULL
        retVal = (TPMT_RSA_DECRYPT *)keyScheme;
    }
    // get here if both the object scheme and the input scheme are
    // not TPM_ALG_NULL. Need to insure that they are the same.
    // IMPLEMENTATION NOTE: This could cause problems if future versions have
    // schemes that have more values than just a hash algorithm. A new function
    // (IsSchemeSame()) might be needed then.
    else if(   keyScheme->scheme == scheme->scheme
            && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
    {
        retVal = scheme;
    }
    // two different, incompatible schemes specified will return NULL
    return retVal;
}
```

11.2.6.6 CryptDecryptRSA()

This function is the interface to _cpri__DecryptRSA(). It handles the return codes from that function and converts them from CRYPT_RESULT to TPM_RC values. The *rsaKey* parameter must reference an RSA decryption key.

Table 132

Error Returns	Meaning
TPM_RC_BINDING	Public and private parts of the key are not cryptographically bound.
TPM_RC_SIZE	Size of data to decrypt is not the same as the key size.
TPM_RC_VALUE	Numeric value of the encrypted data is greater than the public exponent, or output buffer is too small for the decrypted message.

```
TPM_RC
CryptDecryptRSA(
    UINT16              *dataOutSize,    // OUT: size of plain text in bytes
    BYTE                *dataOut,        // OUT: plain text
    OBJECT              *rsaKey,         // IN: internal RSA key
    TPMT_RSA_DECRYPT    *scheme,         // IN: selects the padding scheme
    UINT16               cipherInSize,   // IN: size of cipher text  in byte
    BYTE                *cipherIn,       // IN: cipher text
    const char          *label           // IN: a label, when needed
    )
```

```
{
    RSA_KEY         key;
    CRYPT_RESULT    retVal = CRYPT_SUCCESS;
    UINT32          dSize;                      // Place to put temporary value for the
                                                // returned data size
    TPMI_ALG_HASH   hashAlg = TPM_ALG_NULL; // hash algorithm in the selected
                                                // padding scheme
    TPM_RC          result = TPM_RC_SUCCESS;

    // pointer checks
    pAssert(    (dataOutSize != NULL) && (dataOut != NULL)
            &&  (rsaKey != NULL) && (cipherIn != NULL));

    // The public type is a RSA decrypt key
    pAssert(    (rsaKey->publicArea.type == TPM_ALG_RSA
            &&  rsaKey->publicArea.objectAttributes.decrypt == SET));

    // Must have the private portion loaded.  This check is made before this
    // function is called.
    pAssert(rsaKey->attributes.publicOnly == CLEAR);

    // decryption requires that the private modulus be present
    if(rsaKey->attributes.privateExp == CLEAR)
    {

        // Load key by computing the private exponent
        // CryptLoadPrivateRSA may return TPM_RC_BINDING
        result = CryptLoadPrivateRSA(rsaKey);
    }

    // the input buffer must be the size of the key
    if(result == TPM_RC_SUCCESS)
    {
        if(cipherInSize != rsaKey->publicArea.unique.rsa.t.size)
            result = TPM_RC_SIZE;
        else
        {
            BuildRSA(rsaKey, &key);

            // Initialize the dOutSize parameter
            dSize = *dataOutSize;

            // For OAEP scheme, initialize the hash algorithm for padding
            if(scheme->scheme == TPM_ALG_OAEP)
            {
                hashAlg = scheme->details.oaep.hashAlg;
                TEST_HASH(hashAlg);
            }
            // See if the padding mode needs to be tested
            TEST(scheme->scheme);

            // _cpri__DecryptRSA may return CRYPT_PARAMETER CRYPT_FAIL CRYPT_SCHEME
            retVal = _cpri__DecryptRSA(&dSize, dataOut, &key, scheme->scheme,
                                       cipherInSize, cipherIn, hashAlg, label);

            // Scheme must have been validated when the key was loaded/imported
            pAssert(retVal != CRYPT_SCHEME);

            // Set the return size
            pAssert(dSize <= UINT16_MAX);
            *dataOutSize = (UINT16)dSize;

            // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_FAIL -> TPM_RC_VALUE
            result = TranslateCryptErrors(retVal);
        }
    }
```



```
    return result;
}
```

11.2.6.7 CryptEncryptRSA()

This function provides the interface to _cpri__EncryptRSA(). The object referenced by **rsaKey** is required to be an RSA decryption key.

Table 133

Error Returns	Meaning
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	numeric value of *dataIn* is greater than the key modulus

```
TPM_RC
CryptEncryptRSA(
    UINT16              *cipherOutSize, // OUT: size of cipher text in byte
    BYTE                *cipherOut,     // OUT: cipher text
    OBJECT              *rsaKey,        // IN: internal RSA key
    TPMT_RSA_DECRYPT    *scheme,        // IN: selects the padding scheme
    UINT16               dataInSize,    // IN: size of plain text in byte
    BYTE                *dataIn,        // IN: plain text
    const char          *label          // IN: an optional label
    )
{
    RSA_KEY              key;
    CRYPT_RESULT         retVal;
    UINT32               cOutSize;            // Conversion variable
    TPMI_ALG_HASH        hashAlg = TPM_ALG_NULL;    // hash algorithm in selected
                                                // padding scheme

    // must have a pointer to a key and some data to encrypt
    pAssert(rsaKey != NULL && dataIn != NULL);

    // The public type is a RSA decryption key
    pAssert(   rsaKey->publicArea.type == TPM_ALG_RSA
            && rsaKey->publicArea.objectAttributes.decrypt == SET);

    // If the cipher buffer must be provided and it must be large enough
    // for the result
    pAssert(   cipherOut != NULL
            && cipherOutSize != NULL
            && *cipherOutSize >= rsaKey->publicArea.unique.rsa.t.size);

    // Only need the public key and exponent for encryption
    BuildRSA(rsaKey, &key);

    // Copy the size to the conversion buffer
    cOutSize = *cipherOutSize;

    // For OAEP scheme, initialize the hash algorithm for padding
    if(scheme->scheme == TPM_ALG_OAEP)
    {
        hashAlg = scheme->details.oaep.hashAlg;
        TEST_HASH(hashAlg);
    }

    // This is a public key operation and does not require that the private key
    // be loaded. To verify this, need to do the full algorithm
    TEST(scheme->scheme);

    // Encrypt the data with the public exponent
```

```
    // _cpri__EncryptRSA may return CRYPT_PARAMETER or CRYPT_SCHEME
    retVal = _cpri__EncryptRSA(&cOutSize,cipherOut, &key, scheme->scheme,
                               dataInSize, dataIn, hashAlg, label);

    pAssert (cOutSize <= UINT16_MAX);
    *cipherOutSize = (UINT16)cOutSize;
    // CRYPT_PARAMETER -> TPM_RC_VALUE, CRYPT_SCHEME -> TPM_RC_SCHEME
    return TranslateCryptErrors(retVal);
}
```

11.2.6.8 CryptSignRSA()

This function is used to sign a digest with an RSA signing key.

Table 134

Error Returns	Meaning
TPM_RC_BINDING	public and private part of *signKey* are not properly bound
TPM_RC_SCHEME	*scheme* is not supported
TPM_RC_VALUE	*hashData* is larger than the modulus of *signKey*, or the size of *hashData* does not match hash algorithm in *scheme*


```
static TPM_RC
CryptSignRSA(
    OBJECT              *signKey,       // IN: RSA key signs the hash
    TPMT_SIG_SCHEME     *scheme,        // IN: sign scheme
    TPM2B_DIGEST        *hashData,      // IN: hash to be signed
    TPMT_SIGNATURE      *sig            // OUT: signature
    )
{
    UINT32               signSize;
    RSA_KEY              key;
    CRYPT_RESULT         retVal;
    TPM_RC               result = TPM_RC_SUCCESS;

    pAssert(    (signKey != NULL) && (scheme != NULL)
             && (hashData != NULL) && (sig != NULL));

    // assume that the key has private part loaded and that it is a signing key.
    pAssert(   (signKey->attributes.publicOnly == CLEAR)
            && (signKey->publicArea.objectAttributes.sign == SET));

    // check if the private exponent has been computed
    if(signKey->attributes.privateExp == CLEAR)
        // May return TPM_RC_BINDING
        result = CryptLoadPrivateRSA(signKey);

    if(result == TPM_RC_SUCCESS)
    {
        BuildRSA(signKey, &key);

        // Make sure that the hash is tested
        TEST_HASH(sig->signature.any.hashAlg);

        // Run a test of the RSA sign
        TEST(scheme->scheme);

        // _crypi__SignRSA can return CRYPT_SCHEME and CRYPT_PARAMETER
        retVal = _cpri__SignRSA(&signSize,
                                sig->signature.rsassa.sig.t.buffer,
                                &key,
```

```
                                sig->sigAlg,
                                sig->signature.any.hashAlg,
                                hashData->t.size, hashData->t.buffer);
        pAssert(signSize <= UINT16_MAX);
        sig->signature.rsassa.sig.t.size = (UINT16)signSize;

        // CRYPT_SCHEME -> TPM_RC_SCHEME; CRYPT_PARAMTER -> TPM_RC_VALUE
        result = TranslateCryptErrors(retVal);
    }
    return result;
}
```

11.2.6.9 CryptRSAVerifySignature()

This function is used to verify signature signed by a RSA key.

Table 135

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not genuine
TPM_RC_SCHEME	signature scheme not supported

```
static TPM_RC
CryptRSAVerifySignature(
    OBJECT          *signKey,       // IN: RSA key signed the hash
    TPM2B_DIGEST    *digestData,    // IN: digest being signed
    TPMT_SIGNATURE  *sig            // IN: signature to be verified
    )
{
    RSA_KEY              key;
    CRYPT_RESULT         retVal;
    TPM_RC               result;

    // Validate parameter assumptions
    pAssert((signKey != NULL) && (digestData != NULL) && (sig != NULL));

    TEST_HASH(sig->signature.any.hashAlg);
    TEST(sig->sigAlg);

    // This is a public-key-only operation
    BuildRSA(signKey, &key);

    // Call crypto engine to verify signature
    // _cpri_ValidateSignaturRSA may return CRYPT_FAIL or CRYPT_SCHEME
    retVal = _cpri__ValidateSignatureRSA(&key,
                                         sig->sigAlg,
                                         sig->signature.any.hashAlg,
                                         digestData->t.size,
                                         digestData->t.buffer,
                                         sig->signature.rsassa.sig.t.size,
                                         sig->signature.rsassa.sig.t.buffer,
                                         0);
    // _cpri__ValidateSignatureRSA can return CRYPT_SUCCESS, CRYPT_FAIL, or
    // CRYPT_SCHEME. Translate CRYPT_FAIL to TPM_RC_SIGNATURE
    if(retVal == CRYPT_FAIL)
        result = TPM_RC_SIGNATURE;
    else
        // CRYPT_SCHEME -> TPM_RC_SCHEME
        result = TranslateCryptErrors(retVal);

    return result;
}
```

```
#endif //TPM_ALG_RSA      //% 2
```

11.2.7 ECC Functions

11.2.7.1 CryptEccGetCurveDataPointer()

This function returns a pointer to an ECC_CURVE_VALUES structure that contains the parameters for the key size and schemes for a given curve.

```
#ifdef TPM_ALG_ECC //% 3
static const ECC_CURVE    *
CryptEccGetCurveDataPointer(
    TPM_ECC_CURVE    curveID        // IN: id of the curve
    )
{
    return _cpri__EccGetParametersByCurveId(curveID);
}
```

11.2.7.2 CryptEccGetKeySizeInBits()

This function returns the size in bits of the key associated with a curve.

```
UINT16
CryptEccGetKeySizeInBits(
    TPM_ECC_CURVE    curveID        // IN: id of the curve
    )
{
    const ECC_CURVE          *curve = CryptEccGetCurveDataPointer(curveID);
    UINT16                    keySizeInBits = 0;

    if(curve != NULL)
        keySizeInBits = curve->keySizeBits;

    return keySizeInBits;
}
```

11.2.7.3 CryptEccGetKeySizeBytes()

This macro returns the size of the ECC key in bytes. It uses CryptEccGetKeySizeInBits().

```
// The next lines will be placed in CyrptUtil_fp.h with the //% removed
//% #define CryptEccGetKeySizeInBytes(curve)              \
//%             ((CryptEccGetKeySizeInBits(curve)+7)/8)
```

11.2.7.4 CryptEccGetParameter()

This function returns a pointer to an ECC curve parameter. The parameter is selected by a single character designator from the set of {pnabxyh}.

```
LIB_EXPORT const TPM2B *
CryptEccGetParameter(
    char             p,             // IN: the parameter selector
    TPM_ECC_CURVE    curveId        // IN: the curve id
    )
{
    const ECC_CURVE     *curve = _cpri__EccGetParametersByCurveId(curveId);
    const TPM2B         *parameter = NULL;

    if(curve != NULL)
```

```
    {
        switch (p)
        {
        case 'p':
            parameter = curve->curveData->p;
            break;
        case 'n':
            parameter =  curve->curveData->n;
            break;
        case 'a':
            parameter =  curve->curveData->a;
            break;
        case 'b':
            parameter =  curve->curveData->b;
            break;
        case 'x':
            parameter =  curve->curveData->x;
            break;
        case 'y':
            parameter =  curve->curveData->y;
            break;
        case 'h':
            parameter =  curve->curveData->h;
            break;
        default:
            break;
        }
    }
    return parameter;
}
```

11.2.7.5 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```
const TPMT_ECC_SCHEME *
CryptGetCurveSignScheme(
    TPM_ECC_CURVE     curveId       // IN: The curve selector
    )
{
    const ECC_CURVE          *curve = _cpri__EccGetParametersByCurveId(curveId);
    const TPMT_ECC_SCHEME    *scheme = NULL;

    if(curve != NULL)
        scheme =  &(curve->sign);
    return scheme;
}
```

11.2.7.6 CryptEccIsPointOnCurve()

This function will validate that an ECC point is on the curve of given *curveID*.

Table 136

Return Value	Meaning
TRUE	if the point is on curve
FALSE	if the point is not on curve

```
BOOL
CryptEccIsPointOnCurve(
```

```
    TPM_ECC_CURVE    curveID,        // IN: ECC curve ID
    TPMS_ECC_POINT  *Q               // IN: ECC point
    )
{
    // Make sure that point multiply is working
    TEST(TPM_ALG_ECC);
    // Check point on curve logic by seeing if the test key is on the curve

    // Call crypto engine function to check if a ECC public point is on the
    // given curve
    if(_cpri__EccIsPointOnCurve(curveID, Q))
        return TRUE;
    else
        return FALSE;
}
```

11.2.7.7 CryptNewEccKey()

This function creates a random ECC key that is not derived from other parameters as is a Primary Key.

```
TPM_RC
CryptNewEccKey(
    TPM_ECC_CURVE            curveID,        // IN: ECC curve
    TPMS_ECC_POINT          *publicPoint,    // OUT: public point
    TPM2B_ECC_PARAMETER     *sensitive       // OUT: private area
    )
{
    TPM_RC               result = TPM_RC_SUCCESS;
    // _cpri__GetEphemeralECC may return CRYPT_PARAMETER
    if(_cpri__GetEphemeralEcc(publicPoint, sensitive, curveID) != CRYPT_SUCCESS)
        // Something is wrong with the key.
        result = TPM_RC_KEY;

    return result;
}
```

11.2.7.8 CryptEccPointMultiply()

This function is used to perform a point multiply $R = [d]Q$. If Q is not provided, the multiplication is performed using the generator point of the curve.

Table 137

xError Returns	Meaning
TPM_RC_ECC_POINT	invalid optional ECC point *pIn*
TPM_RC_NO_RESULT	multiplication resulted in a point at infinity
TPM_RC_CANCELED	if a self-test was done, it might have been aborted

```
TPM_RC
CryptEccPointMultiply(
    TPMS_ECC_POINT          *pOut,           // OUT: output point
    TPM_ECC_CURVE            curveId,        // IN: curve selector
    TPM2B_ECC_PARAMETER     *dIn,            // IN: public scalar
    TPMS_ECC_POINT          *pIn             // IN: optional point
    )
{
    TPM2B_ECC_PARAMETER     *n = NULL;
    CRYPT_RESULT            retVal;

```



```
    pAssert(pOut != NULL && dIn != NULL);

    if(pIn != NULL)
    {
        n = dIn;
        dIn = NULL;
    }
    // Do a test of point multiply
    TEST(TPM_ALG_ECC);

    // _cpri__EccPointMultiply may return CRYPT_POINT or CRYPT_NO_RESULT
    retVal = _cpri__EccPointMultiply(pOut, curveId, dIn, pIn, n);

    // CRYPT_POINT->TPM_RC_ECC_POINT and CRYPT_NO_RESULT->TPM_RC_NO_RESULT
    return TranslateCryptErrors(retVal);
}
```

11.2.7.9 CryptGenerateKeyECC()

This function generates an ECC key from a seed value.

The method here may not work for objects that have an order *(G)* that with a different size than a private key.

Table 138

Error Returns	Meaning
TPM_RC_VALUE	hash algorithm is not supported

```
static TPM_RC
CryptGenerateKeyECC(
    TPMT_PUBLIC      *publicArea,     // IN/OUT: The public area template for the new
                                      //         key.
    TPMT_SENSITIVE   *sensitive,      // IN/OUT: the sensitive area
    TPM_ALG_ID        hashAlg,        // IN: algorithm for the KDF
    TPM2B_SEED       *seed,           // IN: the seed value
    TPM2B_NAME       *name,           // IN: the name of the object
    UINT32           *counter         // OUT: the iteration counter
    )
{
    CRYPT_RESULT          retVal;

    TEST_HASH(hashAlg);
    TEST(ALG_ECDSA_VALUE); // ECDSA is used to verify each key

    // The iteration counter has no meaning for ECC key generation. The parameter
    // will be overloaded for those implementations that have a requirement for
    // doing pair-wise consistency checks on signing keys. If the counter parameter
    // is 0 or NULL, then no consistency check is done. If it is other than 0, then
    // a consistency check is run. This modification allow this code to work with
    // the existing versions of the CrytpoEngine and with FIPS-compliant versions
    // as well.
    *counter = (UINT32)(publicArea->objectAttributes.sign == SET);

    // _cpri__GenerateKeyEcc only has one error return (CRYPT_PARAMETER) which means
    // that the hash algorithm is not supported. This should not be possible
    retVal = _cpri__GenerateKeyEcc(&publicArea->unique.ecc,
                                   &sensitive->sensitive.ecc,
                                   publicArea->parameters.eccDetail.curveID,
                                   hashAlg, &seed->b, "ECC key by vendor",
                                   &name->b, counter);
    // This will only be useful if _cpri__GenerateKeyEcc return CRYPT_CANCEL
    return TranslateCryptErrors(retVal);
```

```
}
```

11.2.7.10 CryptSignECC()

This function is used for ECC signing operations. If the signing scheme is a split scheme, and the signing operation is successful, the commit value is retired.

Table 139

Error Returns	Meaning
TPM_RC_SCHEME	unsupported *scheme*
TPM_RC_VALUE	invalid commit status (in case of a split scheme) or failed to generate r value.

```
static TPM_RC
CryptSignECC(
    OBJECT                  *signKey,       // IN: ECC key to sign the hash
    TPMT_SIG_SCHEME         *scheme,        // IN: sign scheme
    TPM2B_DIGEST            *hashData,      // IN: hash to be signed
    TPMT_SIGNATURE          *signature      // OUT: signature
    )
{
    TPM2B_ECC_PARAMETER      r;
    TPM2B_ECC_PARAMETER     *pr = NULL;
    CRYPT_RESULT             retVal;

    // Run a test of the ECC sign and verify if it has not already been run
    TEST_HASH(scheme->details.any.hashAlg);
    TEST(scheme->scheme);

    if(CryptIsSplitSign(scheme->scheme))
    {
        // When this code was written, the only split scheme was ECDAA
        // (which can also be used for U-Prove).
        if(!CryptGenerateR(&r,
                           &scheme->details.ecdaa.count,
                           signKey->publicArea.parameters.eccDetail.curveID,
                           &signKey->name))
            return TPM_RC_VALUE;
        pr = &r;
    }
    // Call crypto engine function to sign
    // _cpri__SignEcc may return CRYPT_SCHEME
    retVal = _cpri__SignEcc(&signature->signature.ecdsa.signatureR,
                            &signature->signature.ecdsa.signatureS,
                            scheme->scheme,
                            scheme->details.any.hashAlg,
                            signKey->publicArea.parameters.eccDetail.curveID,
                            &signKey->sensitive.sensitive.ecc,
                            &hashData->b,
                            pr
                            );
    if(CryptIsSplitSign(scheme->scheme) && retVal == CRYPT_SUCCESS)
        CryptEndCommit(scheme->details.ecdaa.count);
    // CRYPT_SCHEME->TPM_RC_SCHEME
    return TranslateCryptErrors(retVal);
}
```


11.2.7.11 CryptECCVerifySignature()

This function is used to verify a signature created with an ECC key.

Table 140

Error Returns	Meaning
TPM_RC_SIGNATURE	if signature is not valid
TPM_RC_SCHEME	the signing scheme or *hashAlg* is not supported

```
static TPM_RC
CryptECCVerifySignature(
    OBJECT          *signKey,       // IN: ECC key signed the hash
    TPM2B_DIGEST    *digestData,    // IN: digest being signed
    TPMT_SIGNATURE  *signature      // IN: signature to be verified
    )
{
    CRYPT_RESULT        retVal;

    TEST_HASH(signature->signature.any.hashAlg);
    TEST(signature->sigAlg);

    // This implementation uses the fact that all the defined ECC signing
    // schemes have the hash as the first parameter.
    // _cpriValidateSignatureEcc may return CRYPT_FAIL or CRYP_SCHEME
    retVal = _cpri__ValidateSignatureEcc(&signature->signature.ecdsa.signatureR,
                                  &signature->signature.ecdsa.signatureS,
                                  signature->sigAlg,
                                  signature->signature.any.hashAlg,
                                  signKey->publicArea.parameters.eccDetail.curveID,
                                  &signKey->publicArea.unique.ecc,
                                  &digestData->b);
    if(retVal == CRYPT_FAIL)
        return TPM_RC_SIGNATURE;
    // CRYPT_SCHEME->TPM_RC_SCHEME
    return TranslateCryptErrors(retVal);
}
```


11.2.7.12 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit(). If *c* is not NULL, the TPM will validate that the *gr.commitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Table 141

Return Value	Meaning
TRUE	r value computed
FALSE	no r value computed

```
BOOL
CryptGenerateR(
    TPM2B_ECC_PARAMETER     *r,             // OUT: the generated random value
    UINT16                  *c,             // IN/OUT: count value.
    TPMI_ECC_CURVE           curveID,       // IN: the curve for the value
    TPM2B_NAME              *name           // IN: optional name of a key to
```

```
                                              //     associate with 'r'
    )
{
    // This holds the marshaled g_commitCounter.
    TPM2B_TYPE(8B, 8);
    TPM2B_8B                cntr = {8,{0}};

    UINT32                   iterations;
    const TPM2B             *n;
    UINT64                   currentCount = gr.commitCounter;
    // This is just to suppress a compiler warning about a conditional expression
    // being a constant. This is because of the macro expansion of ryptKDFa
    TPMI_ALG_HASH            hashAlg = CONTEXT_INTEGRITY_HASH_ALG;

    n =  CryptEccGetParameter('n', curveID);
    pAssert(r != NULL && n != NULL);

    // If this is the commit phase, use the current value of the commit counter
    if(c != NULL)
    {

        UINT16      t1;
        // if the array bit is not set, can't use the value.
        if(!BitIsSet((*c & COMMIT_INDEX_MASK), gr.commitArray,
                     sizeof(gr.commitArray)))
            return FALSE;

        // If it is the sign phase, figure out what the counter value was
        // when the commitment was made.
        //
        // When gr.commitArray has less than 64K bits, the extra
        // bits of 'c' are used as a check to make sure that the
        // signing operation is not using an out of range count value
        t1 = (UINT16)currentCount;

        // If the lower bits of c are greater or equal to the lower bits of t1
        // then the upper bits of t1 must be one more than the upper bits
        // of c
        if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
            // Since the counter is behind, reduce the current count
            currentCount = currentCount - (COMMIT_INDEX_MASK + 1);

        t1 = (UINT16)currentCount;
        if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
            return FALSE;
        // set the counter to the value that was
        // present when the commitment was made
        currentCount = (currentCount & 0xffffffffffff0000) | *c;

    }
    // Marshal the count value to a TPM2B buffer for the KDF
    cntr.t.size = sizeof(currentCount);
    UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);

    // Now can do the KDF to create the random value for the signing operation
    // During the creation process, we may generate an r that does not meet the
    // requirements of the random value.
    // want to generate a new r.

    r->t.size = n->size;

    // Arbitrary upper limit on the number of times that we can look for
    // a suitable random value.  The normally number of tries will be 1.
    for(iterations = 1; iterations < 1000000;)
    {
        BYTE     *pr = &r->b.buffer[0];
```



```
        int      i;
        CryptKDFa(hashAlg, &gr.commitNonce.b, "ECDAA Commit",
                  name, &cntr.b, n->size * 8, r->t.buffer, &iterations);

        // random value must be less than the prime
        if(CryptCompare(r->b.size, r->b.buffer, n->size, n->buffer) >= 0)
            continue;

        // in this implementation it is required that at least bit
        // in the upper half of the number be set
        for(i = n->size/2; i > 0; i--)
            if(*pr++ != 0)
                return TRUE;
    }
    return FALSE;
}
```

11.2.7.13 CryptCommit()

This function is called when the count value is committed. The gr.*commitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```
UINT16
CryptCommit(
    void
    )
{
    UINT16      oldCount = (UINT16)gr.commitCounter;
    gr.commitCounter++;
    BitSet(oldCount & COMMIT_INDEX_MASK, gr.commitArray, sizeof(gr.commitArray));
    return oldCount;
}
```

11.2.7.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the gr.*commitArray* bit associated with the count value so that it can't be used again.

```
void
CryptEndCommit(
    UINT16          c                // IN: the counter value of the commitment
    )
{
    BitClear((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
}
```

11.2.7.15 CryptCommitCompute()

This function performs the computations for the TPM2_Commit() command. This could be a macro.

Table 142

Error Returns	Meaning
TPM_RC_NO_RESULT	*K*, *L*, or *E* is the point at infinity
TPM_RC_CANCELLED	command was canceled

```
TPM_RC
```

```
CryptCommitCompute(
    TPMS_ECC_POINT          *K,            // OUT: [d]B
    TPMS_ECC_POINT          *L,            // OUT: [r]B
    TPMS_ECC_POINT          *E,            // OUT: [r]M
    TPM_ECC_CURVE            curveID,      // IN: The curve for the computations
    TPMS_ECC_POINT          *M,            // IN: M (P1)
    TPMS_ECC_POINT          *B,            // IN: B (x2, y2)
    TPM2B_ECC_PARAMETER     *d,            // IN: the private scalar
    TPM2B_ECC_PARAMETER     *r             // IN: the computed r value
    )
{
    TEST(ALG_ECDH_VALUE);
    // CRYPT_NO_RESULT->TPM_RC_NO_RESULT CRYPT_CANCEL->TPM_RC_CANCELLED
    return TranslateCryptErrors(
               _cpri__EccCommitCompute(K, L , E, curveID, M, B, d, r));
}
```

11.2.7.16 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

Table 143

Return Value	Meaning
TRUE	Get parameters success
FALSE	Unsupported ECC curve ID

```
BOOL
CryptEccGetParameters(
    TPM_ECC_CURVE                curveId,       // IN: ECC curve ID
    TPMS_ALGORITHM_DETAIL_ECC   *parameters     // OUT: ECC parameters
    )
{
    const ECC_CURVE             *curve = _cpri__EccGetParametersByCurveId(curveId);
    const ECC_CURVE_DATA        *data;
    BOOL                         found = curve != NULL;

    if(found)
    {

        data = curve->curveData;

        parameters->curveID = curve->curveId;

        // Key size in bit
        parameters->keySize = curve->keySizeBits;

        // KDF
        parameters->kdf = curve->kdf;

        // Sign
        parameters->sign = curve->sign;

        // Copy p value
        MemoryCopy2B(&parameters->p.b, data->p, sizeof(parameters->p.t.buffer));

        // Copy a value
        MemoryCopy2B(&parameters->a.b, data->a, sizeof(parameters->a.t.buffer));

        // Copy b value
        MemoryCopy2B(&parameters->b.b, data->b, sizeof(parameters->b.t.buffer));

```



```
        // Copy Gx value
        MemoryCopy2B(&parameters->gX.b, data->x, sizeof(parameters->gX.t.buffer));

        // Copy Gy value
        MemoryCopy2B(&parameters->gY.b, data->y, sizeof(parameters->gY.t.buffer));

        // Copy n value
        MemoryCopy2B(&parameters->n.b, data->n, sizeof(parameters->n.t.buffer));

        // Copy h value
        MemoryCopy2B(&parameters->h.b, data->h, sizeof(parameters->h.t.buffer));
    }
    return found;
}
#if CC_ZGen_2Phase == YES
```

CryptEcc2PhaseKeyExchange() This is the interface to the key exchange function.

```
TPM_RC
CryptEcc2PhaseKeyExchange(
    TPMS_ECC_POINT          *outZ1,         // OUT: the computed point
    TPMS_ECC_POINT          *outZ2,         // OUT: optional second point
    TPM_ALG_ID               scheme,        // IN: the key exchange scheme
    TPM_ECC_CURVE            curveId,       // IN: the curve for the computations
    TPM2B_ECC_PARAMETER     *dsA,           // IN: static private TPM key
    TPM2B_ECC_PARAMETER     *deA,           // IN: ephemeral private TPM key
    TPMS_ECC_POINT          *QsB,           // IN: static public party B key
    TPMS_ECC_POINT          *QeB            // IN: ephemeral public party B key
    )
{
    return (TranslateCryptErrors(_cpri__C_2_2_KeyExchange(outZ1,
                                                          outZ2,
                                                          scheme,
                                                          curveId,
                                                          dsA,
                                                          deA,
                                                          QsB,
                                                          QeB)));
}
#endif //  CC_ZGen_2Phase
#endif //TPM_ALG_ECC  //% 3
```

11.2.7.17 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme The only anonymous scheme is ECDAA. ECDAA can be used to do things like U-Prove.

```
BOOL
CryptIsSchemeAnonymous(
    TPM_ALG_ID        scheme         // IN: the scheme algorithm to test
    )
{
#ifdef TPM_ALG_ECDAA
    return (scheme == TPM_ALG_ECDAA);
#else
    UNREFERENCED(scheme);
    return  0;
#endif
}
```

11.2.8 Symmetric Functions

11.2.8.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```
void
ParmDecryptSym(
    TPM_ALG_ID       symAlg,        // IN: the symmetric algorithm
    TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
    UINT16           keySizeInBits, // IN: key key size in bits
    TPM2B           *key,           // IN: KDF HMAC key
    TPM2B           *nonceCaller,   // IN: nonce caller
    TPM2B           *nonceTpm,      // IN: nonce TPM
    UINT32           dataSize,      // IN: size of parameter buffer
    BYTE            *data           // OUT: buffer to be decrypted
    )
{
    // KDF output buffer
    // It contains parameters for the CFB encryption
    // From MSB to LSB, they are the key and iv
    BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
    // Symmetric key size in byte
    UINT16           keySize = (keySizeInBits + 7) / 8;
    TPM2B_IV         iv;

    iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
    // If there is decryption to do...
    if(iv.t.size > 0)
    {
        // Generate key and iv
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
        // label values.
        CryptKDFa(hash, key, "CFB", nonceCaller, nonceTpm,
                  keySizeInBits + (iv.t.size * 8), symParmString, NULL);
        MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
                   sizeof(iv.t.buffer));

        CryptSymmetricDecrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
                              symParmString, &iv, dataSize, data);
    }
    return;
}
```

11.2.8.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
void
ParmEncryptSym(
    TPM_ALG_ID       symAlg,        // IN: symmetric algorithm
    TPM_ALG_ID       hash,          // IN: hash algorithm for KDFa
    UINT16           keySizeInBits, // IN: AES key size in bits
    TPM2B           *key,           // IN: KDF HMAC key
    TPM2B           *nonceCaller,   // IN: nonce caller
    TPM2B           *nonceTpm,      // IN: nonce TPM
    UINT32           dataSize,      // IN: size of parameter buffer
    BYTE            *data           // OUT: buffer to be encrypted
    )
{
    // KDF output buffer
    // It contains parameters for the CFB encryption
    BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
```



```

    // Symmetric key size in bytes
    UINT16          keySize = (keySizeInBits + 7) / 8;

    TPM2B_IV        iv;

    iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
    // See if there is any encryption to do
    if(iv.t.size > 0)
    {
        // Generate key and iv
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
        // label values.
        CryptKDFa(hash, key, "CFB", nonceTpm, nonceCaller,
                  keySizeInBits + (iv.t.size * 8), symParmString, NULL);

        MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size,
                   sizeof(iv.t.buffer));

        CryptSymmetricEncrypt(data, symAlg, keySizeInBits, TPM_ALG_CFB,
                              symParmString, &iv, dataSize, data);
    }
    return;
}
```

11.2.8.3 CryptGenerateNewSymmetric()

This function creates the sensitive symmetric values for an HMAC or symmetric key. If the sensitive area is zero, then the sensitive creation key data is copied. If it is not zero, then the TPM will generate a random value of the selected size.

```
void
CryptGenerateNewSymmetric(
    TPMS_SENSITIVE_CREATE    *sensitiveCreate,    // IN: sensitive creation data
    TPMT_SENSITIVE           *sensitive,          // OUT: sensitive area
    TPM_ALG_ID                hashAlg,            // IN: hash algorithm for the KDF
    TPM2B_SEED               *seed,               // IN: seed used in creation
    TPM2B_NAME               *name                // IN: name of the object
    )
{
    // This function is called to create a key and obfuscation value for a
    // symmetric key that can either be a block cipher or an XOR key. The buffer
    // in sensitive->sensitive will hold either. When we call the function
    // to copy the input value or generated value to the sensitive->sensitive
    // buffer we will need to have a size for the output buffer. This define
    // computes the maximum that it might need to be and uses that. It will always
    // be smaller than the largest value that will fit.
    #define MAX_SENSITIVE_SIZE                                                \
        (MAX(sizeof(sensitive->sensitive.bits.t.buffer),                      \
            sizeof(sensitive->sensitive.sym.t.buffer)))

    // set the size of the obfuscation value
    sensitive->seedValue.t.size = CryptGetHashDigestSize(hashAlg);

    // If the input sensitive size is zero, then create both the sensitive data
    // and the obfuscation value
    if(sensitiveCreate->data.t.size == 0)
    {
        BYTE                        symValues[MAX_SYM_KEY_BYTES +  MAX_DIGEST_SIZE];
        INT16                       requestSize;

        // Set the size of the request to be the size of the key and the
        // obfuscation value
        requestSize =   sensitive->sensitive.sym.t.size
```

```
                           + sensitive->seedValue.t.size;

        _cpri__GenerateSeededRandom(requestSize, symValues, hashAlg, &seed->b,
                                    "symmetric sensitive", &name->b, NULL);

        // Copy the new key
        MemoryCopy(sensitive->sensitive.sym.t.buffer,
                   symValues, sensitive->sensitive.sym.t.size,
                   MAX_SENSITIVE_SIZE);

        // copy the obfuscation value
        MemoryCopy(sensitive->seedValue.t.buffer,
                   &symValues[sensitive->sensitive.sym.t.size],
                   sensitive->seedValue.t.size,
                   sizeof(sensitive->seedValue.t.buffer));
    }
    else
    {
        // Copy input symmetric key to sensitive area as long as it will fit
        MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
                     MAX_SENSITIVE_SIZE);

        // Create the obfuscation value
        _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
                                    sensitive->seedValue.t.buffer,
                                    hashAlg, &seed->b,
                                    "symmetric obfuscation", &name->b, NULL);
    }
    return;
}
```

11.2.8.4 CryptGenerateKeySymmetric()

This function derives a symmetric cipher key from the provided seed.

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area

```
static TPM_RC
CryptGenerateKeySymmetric(
    TPMT_PUBLIC            *publicArea,        // IN/OUT: The public area template
                                               //     for the new key.
    TPMS_SENSITIVE_CREATE  *sensitiveCreate,   // IN: sensitive creation data
    TPMT_SENSITIVE         *sensitive,         // OUT: sensitive area
    TPM_ALG_ID              hashAlg,           // IN: hash algorithm for the KDF
    TPM2B_SEED             *seed,              // IN: seed used in creation
    TPM2B_NAME             *name               // IN: name of the object
    )
{
    // If this is not a new key, then the provided key data must be the right size
    if(publicArea->objectAttributes.sensitiveDataOrigin == CLEAR)
    {
        if(     (sensitiveCreate->data.t.size * 8)
            !=  publicArea->parameters.symDetail.sym.keyBits.sym)
            return TPM_RC_KEY_SIZE;
        // Make sure that the key size is OK.
        // This implementation only supports symmetric key sizes that are
        // multiples of 8
        if(publicArea->parameters.symDetail.sym.keyBits.sym % 8 != 0)
            return TPM_RC_KEY_SIZE;
    }
    else
```



```
    {
        // TPM is going to generate the key so set the size
        sensitive->sensitive.sym.t.size
            = publicArea->parameters.symDetail.sym.keyBits.sym / 8;
        sensitiveCreate->data.t.size = 0;
    }
    // Fill in the sensitive area
    CryptGenerateNewSymmetric(sensitiveCreate, sensitive, hashAlg,
                              seed, name);

    // Create unique area in public
    CryptComputeSymmetricUnique(publicArea->nameAlg,
                                sensitive, &publicArea->unique.sym);

    return TPM_RC_SUCCESS;
}
```

11.2.8.5 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM_RC_SUCCESS.

```
#ifdef TPM_ALG_KEYEDHASH //% 5
void
CryptXORObfuscation(
    TPM_ALG_ID       hash,          // IN: hash algorithm for KDF
    TPM2B           *key,           // IN: KDF key
    TPM2B           *contextU,      // IN: contextU
    TPM2B           *contextV,      // IN: contextV
    UINT32           dataSize,      // IN: size of data buffer
    BYTE            *data           // IN/OUT: data to be XORed in place
    )
{
    BYTE             mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
    BYTE            *pm;
    UINT32           i;
    UINT32           counter = 0;
    UINT16           hLen = CryptGetHashDigestSize(hash);
    UINT32           requestSize = dataSize * 8;
    INT32            remainBytes = (INT32) dataSize;

    pAssert((key != NULL) && (data != NULL) && (hLen != 0));

    // Call KDFa to generate XOR mask
    for(; remainBytes > 0; remainBytes -= hLen)
    {
        // Make a call to KDFa to get next iteration
        // See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for normative KDF
        // label values.
        CryptKDFaOnce(hash, key, "XOR", contextU, contextV,
                      requestSize, mask, &counter);

        // XOR next piece of the data
        pm = mask;
        for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
            *data++ ^= *pm++;
    }
    return;
}
#endif //TPM_ALG_KEYED_HASH //%5
```

11.2.9 Initialization and shut down

11.2.9.1 CryptInitUnits()

This function is called when the TPM receives a _TPM_Init() indication. After function returns, the hash algorithms should be available.

NOTE The hash algorithms do not have to be tested, they just need to be available. They have to be tested before the TPM can accept HMAC authorization or return any result that relies on a hash algorithm.

```
void
CryptInitUnits(
    void
    )
{
    // Initialize the vector of implemented algorithms
    AlgorithmGetImplementedVector(&g_implementedAlgorithms);

    // Indicate that all test are necessary
    CryptInitializeToTest();

    // Call crypto engine unit initialization
    // It is assumed that crypt engine initialization should always succeed.
    // Otherwise, TPM should go to failure mode.
    if(_cpri__InitCryptoUnits(&TpmFail) != CRYPT_SUCCESS)
        FAIL(FATAL_ERROR_INTERNAL);
    return;
}
```

11.2.9.2 CryptStopUnits()

This function is only used in a simulated environment. There should be no reason to shut down the cryptography on an actual TPM other than loss of power. After receiving TPM2_Startup(), the TPM should be able to accept commands until it loses power and, unless the TPM is in Failure Mode, the cryptographic algorithms should be available.

```
void
CryptStopUnits(
    void
    )
{
    // Call crypto engine unit stopping
    _cpri__StopCryptoUnits();

    return;
}
```

11.2.9.3 CryptUtilStartup()

This function is called by TPM2_Startup() to initialize the functions in this crypto library and in the provided CryptoEngine(). In this implementation, the only initialization required in this library is initialization of the Commit nonce on TPM Reset.

This function returns false if some problem prevents the functions from starting correctly. The TPM should go into failure mode.

```
BOOL
CryptUtilStartup(
    STARTUP_TYPE     type             // IN: the startup type
    )
```



```
{
    // Make sure that the crypto library functions are ready.
    // NOTE: need to initialize the crypto before loading
    // the RND state may trigger a self-test which
    // uses the
    if( !_cpri__Startup())
        return FALSE;

    // Initialize the state of the RNG.
    CryptDrbgGetPutState(PUT_STATE);

    if(type == SU_RESET)
    {
#ifdef TPM_ALG_ECC
        // Get a new  random commit nonce
        gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
        _cpri__GenerateRandom(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
        // Reset the counter and commit array
        gr.commitCounter = 0;
        MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
#endif // TPM_ALG_ECC
    }

    // If the shutdown was orderly, then the values recovered from NV will
    // be OK to use. If the shutdown was not orderly, then a TPM Reset was required
    // and we would have initialized in the code above.

    return TRUE;
}
```

11.2.10 Algorithm-Independent Functions

11.2.10.1 Introduction

These functions are used generically when a function of a general type (e.g., symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

11.2.10.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Table 144

Return Value	Meaning
TRUE	if it is an asymmetric algorithm
FALSE	if it is not an asymmetric algorithm

```
BOOL
CryptIsAsymAlgorithm(
    TPM_ALG_ID       algID          // IN: algorithm ID
    )
{
    return (
#ifdef TPM_ALG_RSA
                algID ==  TPM_ALG_RSA
#endif
#if defined TPM_ALG_RSA && defined TPM_ALG_ECC
                ||
#endif
```

```
#ifdef TPM_ALG_ECC
             algID == TPM_ALG_ECC
#endif
           );
}
```

11.2.10.3 CryptGetSymmetricBlockSize()

This function returns the size in octets of the symmetric encryption block used by an algorithm and key size combination.

```
INT16
CryptGetSymmetricBlockSize(
    TPMI_ALG_SYM     algorithm,      // IN: symmetric algorithm
    UINT16           keySize         // IN: key size in bit
    )
{
    return _cpri__GetSymmetricBlockSize(algorithm, keySize);
}
```

11.2.10.4 CryptSymmetricEncrypt()

This function does in-place encryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
void
CryptSymmetricEncrypt(
    BYTE                 *encrypted,     // OUT: the encrypted data
    TPM_ALG_ID            algorithm,     // IN: algorithm for encryption
    UINT16                keySizeInBits, // IN: key size in bits
    TPMI_ALG_SYM_MODE     mode,          // IN: symmetric encryption mode
    BYTE                 *key,           // IN: encryption key
    TPM2B_IV             *ivIn,          // IN/OUT: Input IV and output chaining
                                         //     value for the next block
    UINT32                dataSize,      // IN: data size in byte
    BYTE                 *data           // IN/OUT: data buffer
    )
{
    TPM2B_IV              defaultIv = {0};

    TEST(algorithm);

    pAssert(encrypted != NULL && key != NULL);

    defaultIv.t.size = _cpri__GetSymmetricBlockSize(algorithm, keySizeInBits);

    // If the IV is not provided, or if doing ECB, then use the default IV
    if(ivIn == NULL || mode == TPM_ALG_ECB)
        ivIn = &defaultIv;

    // Make sure that there is an IV and that the provided size is the
    // requried size.
    pAssert(ivIn->t.size == defaultIv.t.size);

   if( _cpri__SymmetricEncrypt(
        encrypted,
        algorithm,
        keySizeInBits,
        key,
        ivIn,
        mode,
        dataSize,
```



```
            data) != CRYPT_SUCCESS)
            FAIL(FATAL_ERROR_CRYPTO);

        return;

}
```

11.2.10.5 CryptSymmetricDecrypt()

This function does in-place decryption of a buffer using the indicated symmetric algorithm, key, IV, and mode. If the symmetric algorithm and mode are not defined, the TPM will fail.

```
void
CryptSymmetricDecrypt(
    BYTE                *decrypted,     // OUT: the decrypted data
    TPM_ALG_ID           algorithm,     // IN: algorithm for encryption
    UINT16               keySizeInBits, // IN: key size in bits
    TPMI_ALG_SYM_MODE    mode,          // IN: symmetric encryption mode
    BYTE                *key,           // IN: encryption key
    TPM2B_IV            *ivIn,          // IN/OUT: Input IV and output chaining
                                        //     value for the next block
    UINT32               dataSize,      // IN: data size in byte
    BYTE                *data           // IN/OUT: data buffer
    )
{
    TPM2B_IV             defaultIv = {0};

    TEST(algorithm);

    pAssert(decrypted != NULL && key != NULL);

    defaultIv.t.size = _cpri__GetSymmetricBlockSize(algorithm, keySizeInBits);

    // If the IV is not provided, or if doing ECB, then use the default IV
    if(ivIn == NULL || mode == TPM_ALG_ECB)
        ivIn = &defaultIv;

    // Make sure that there is an IV and that the provided size is the
    // requried size.
    pAssert(ivIn->t.size == defaultIv.t.size);

    if(_cpri__SymmetricDecrypt(
        decrypted,
        algorithm,
        keySizeInBits,
        key,
        ivIn,
        mode,
        dataSize,
        data) != CRYPT_SUCCESS)
        FAIL(FATAL_ERROR_CRYPTO);

    return;

}
```

11.2.10.6 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_Rewrap() TPM2_MakeCredential(), and TPM2_Duplicate().

Table 145

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```
TPM_RC
CryptSecretEncrypt(
    TPMI_DH_OBJECT          keyHandle,      // IN: encryption key handle
    const char             *label,          // IN: a null-terminated string as L
    TPM2B_DATA             *data,           // OUT: secret value
    TPM2B_ENCRYPTED_SECRET *secret          // OUT: secret structure
    )
{
    TPM_RC        result = TPM_RC_SUCCESS;
    OBJECT       *encryptKey = ObjectGet(keyHandle);  // TPM key used for encrypt

    pAssert(data != NULL && secret != NULL);

    // The output secret value has the size of the digest produced by the nameAlg.
    data->t.size = CryptGetHashDigestSize(encryptKey->publicArea.nameAlg);

    pAssert(encryptKey->publicArea.objectAttributes.decrypt == SET);

    switch(encryptKey->publicArea.type)
    {
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
        {
            TPMT_RSA_DECRYPT            scheme;

            // Use OAEP scheme
            scheme.scheme = TPM_ALG_OAEP;
            scheme.details.oaep.hashAlg = encryptKey->publicArea.nameAlg;

            // Create secret data from RNG
            CryptGenerateRandom(data->t.size, data->t.buffer);

            // Encrypt the data by RSA OAEP into encrypted secret
            result = CryptEncryptRSA(&secret->t.size, secret->t.secret,
                                     encryptKey, &scheme,
                                     data->t.size, data->t.buffer, label);
        }
        break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
        {
            TPMS_ECC_POINT      eccPublic;
            TPM2B_ECC_PARAMETER eccPrivate;
            TPMS_ECC_POINT      eccSecret;
            BYTE                *buffer = secret->t.secret;

            // Need to make sure that the public point of the key is on the
            // curve defined by the key.
            if(!_cpri__EccIsPointOnCurve(
                        encryptKey->publicArea.parameters.eccDetail.curveID,
                        &encryptKey->publicArea.unique.ecc))
```

```
                result = TPM_RC_KEY;
            else
            {

                // Call crypto engine to create an auxiliary ECC key
                // We assume crypt engine initialization should always success.
                // Otherwise, TPM should go to failure mode.
                CryptNewEccKey(encryptKey->publicArea.parameters.eccDetail.curveID,
                               &eccPublic, &eccPrivate);

                // Marshal ECC public to secret structure. This will be used by the
                // recipient to decrypt the secret with their private key.
                secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);

                // Compute ECDH shared secret which is R = [d]Q where d is the
                // private part of the ephemeral key and Q is the public part of a
                // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
                // because the auxiliary ECC key is just created according to the
                // parameters of input ECC encrypt key.
                if(    CryptEccPointMultiply(&eccSecret,
                               encryptKey->publicArea.parameters.eccDetail.curveID,
                               &eccPrivate,
                               &encryptKey->publicArea.unique.ecc)
                   != CRYPT_SUCCESS)
                    result = TPM_RC_KEY;
                else

                    // The secret value is computed from Z using KDFe as:
                    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
                    // Where:
                    //  HashID  the nameAlg of the decrypt key
                    //  Z    the x coordinate (Px) of the product (P) of the point
                    //       (Q) of the secret and the private x coordinate (de,V)
                    //       of the decryption key
                    //  Use a null-terminated string containing "SECRET"
                    //    See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for
                    //    normative KDF label values.
                    //  PartyUInfo  the x coordinate of the point in the secret
                    //              (Qe,U )
                    //  PartyVInfo  the x coordinate of the public key (Qs,V )
                    //  bits    the number of bits in the digest of HashID
                    // Retrieve seed from KDFe

                    CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
                              label, &eccPublic.x.b,
                              &encryptKey->publicArea.unique.ecc.x.b,
                              data->t.size * 8, data->t.buffer);
            }
        }
        break;
#endif //TPM_ALG_ECC

    default:
        FAIL(FATAL_ERROR_INTERNAL);
        break;
    }

    return result;
}
```

11.2.10.7 CryptSecretDecrypt()

Decrypt a secret value by asymmetric (or symmetric) algorithm This function is used for ActivateCredential() and Import for asymmetric decryption, and StartAuthSession() for both asymmetric and symmetric decryption process.

Table 146

Error Returns	Meaning
TPM_RC_ATTRIBUTES	RSA key is not a decryption key
TPM_RC_BINDING	Invalid RSA key (public and private parts are not cryptographically bound.
TPM_RC_ECC_POINT	ECC point in the secret is not on the curve
TPM_RC_INSUFFICIENT	failed to retrieve ECC point from the secret
TPM_RC_NO_RESULT	multiplication resulted in ECC point at infinity
TPM_RC_SIZE	data to decrypt is not of the same size as RSA key
TPM_RC_VALUE	For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For *keyedHash* or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```
TPM_RC
CryptSecretDecrypt(
    TPM_HANDLE                tpmKey,         // IN: decrypt key
    TPM2B_NONCE              *nonceCaller,    // IN: nonceCaller.  It is needed for
                                              //     symmetric decryption.  For
                                              //     asymmetric decryption, this
                                              //     parameter is NULL
    const char               *label,          // IN: a null-terminated string as L
    TPM2B_ENCRYPTED_SECRET   *secret,         // IN: input secret
    TPM2B_DATA               *data            // OUT: decrypted secret value
    )
{
    TPM_RC      result = TPM_RC_SUCCESS;
    OBJECT      *decryptKey = ObjectGet(tpmKey);   //TPM key used for decrypting

    // Decryption for secret
    switch(decryptKey->publicArea.type)
    {

#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
        {
            TPMT_RSA_DECRYPT        scheme;

            // Use OAEP scheme
            scheme.scheme = TPM_ALG_OAEP;
            scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;

            // Set the output buffer capacity
            data->t.size = sizeof(data->t.buffer);

            // Decrypt seed by RSA OAEP
            result = CryptDecryptRSA(&data->t.size, data->t.buffer, decryptKey,
                                     &scheme,
                                     secret->t.size, secret->t.secret,label);
            if(    (result == TPM_RC_SUCCESS)
```



```
               && (data->t.size
                    > CryptGetHashDigestSize(decryptKey->publicArea.nameAlg)))
                result = TPM_RC_VALUE;
        }
        break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
        {
            TPMS_ECC_POINT       eccPublic;
            TPMS_ECC_POINT       eccSecret;
            BYTE                *buffer = secret->t.secret;
            INT32                size = secret->t.size;

            // Retrieve ECC point from secret buffer
            result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
            if(result == TPM_RC_SUCCESS)
            {
                result = CryptEccPointMultiply(&eccSecret,
                               decryptKey->publicArea.parameters.eccDetail.curveID,
                               &decryptKey->sensitive.sensitive.ecc,
                               &eccPublic);

                if(result == TPM_RC_SUCCESS)
                {

                    // Set the size of the "recovered" secret value to be the size
                    // of the digest produced by the nameAlg.
                    data->t.size =
                            CryptGetHashDigestSize(decryptKey->publicArea.nameAlg);

                    // The secret value is computed from Z using KDFe as:
                    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
                    // Where:
                    //  HashID -- the nameAlg of the decrypt key
                    //  Z --  the x coordinate (Px) of the product (P) of the point
                    //        (Q) of the secret and the private x coordinate (de,V)
                    //        of the decryption key
                    //  Use -- a null-terminated string containing "SECRET"
                    //    See ISO/IEC 11889-1, clause 5.4, "KDF Label Parameters" for
                    //    normative KDF label values.
                    //  PartyUInfo -- the x coordinate of the point in the secret
                    //              (Qe,U )
                    //  PartyVInfo -- the x coordinate of the public key (Qs,V )
                    //  bits -- the number of bits in the digest of HashID
                    // Retrieve seed from KDFe
                    CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
                              &eccPublic.x.b,
                              &decryptKey->publicArea.unique.ecc.x.b,
                              data->t.size * 8, data->t.buffer);
                }
            }
        }
        break;
#endif //TPM_ALG_ECC

        case TPM_ALG_KEYEDHASH:
            // The seed size can not be bigger than the digest size of nameAlg
            if(secret->t.size >
                    CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
                result = TPM_RC_VALUE;
            else
            {
                // Retrieve seed by XOR Obfuscation:
                //    seed = XOR(secret, hash, key, nonceCaller, nullNonce)
```

```
                //    where:
                //    secret  the secret parameter from the TPM2_StartAuthHMAC
                //            command
                //            which contains the seed value
                //    hash    nameAlg  of tpmKey
                //    key     the key or data value in the object referenced by
                //            entityHandle in the TPM2_StartAuthHMAC command
                //    nonceCaller the parameter from the TPM2_StartAuthHMAC command
                //    nullNonce   a zero-length nonce
                // XOR Obfuscation in place
                CryptXORObfuscation(decryptKey->publicArea.nameAlg,
                                    &decryptKey->sensitive.sensitive.bits.b,
                                    &nonceCaller->b, NULL,
                                    secret->t.size, secret->t.secret);
                // Copy decrypted seed
                MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
            }
            break;
        case TPM_ALG_SYMCIPHER:
            {
                TPM2B_IV                iv = {0};
                TPMT_SYM_DEF_OBJECT     *symDef;
                // The seed size can not be bigger than the digest size of nameAlg
                if(secret->t.size >
                        CryptGetHashDigestSize(decryptKey->publicArea.nameAlg))
                    result = TPM_RC_VALUE;
                else
                {
                    symDef = &decryptKey->publicArea.parameters.symDetail.sym;
                    iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
                                                           symDef->keyBits.sym);
                    pAssert(iv.t.size != 0);
                    if(nonceCaller->t.size >= iv.t.size)
                        MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size,
                                   sizeof(iv.t.buffer));
                    else
                        MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
                                  nonceCaller->t.size, sizeof(iv.t.buffer));
                    // CFB decrypt in place, using nonceCaller as iv
                    CryptSymmetricDecrypt(secret->t.secret, symDef->algorithm,
                                       symDef->keyBits.sym, TPM_ALG_CFB,
                                       decryptKey->sensitive.sensitive.sym.t.buffer,
                                       &iv, secret->t.size, secret->t.secret);

                    // Copy decrypted seed
                    MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
                }
            }
            break;
        default:
            pAssert(0);
            break;
    }
    return result;
}
```

11.2.10.8 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```
void
CryptParameterEncryption(
    TPM_HANDLE        handle,           // IN: encrypt session handle
    TPM2B            *nonceCaller,      // IN: nonce caller
```

```
    UINT16           leadingSizeInByte, // IN: the size of the leading size field in
                                        //     bytes
    TPM2B_AUTH      *extraKey,          // IN: additional key material other than
                                        //     session auth
    BYTE            *buffer             // IN/OUT: parameter buffer to be encrypted
    )
{
    SESSION     *session = SessionGet(handle);  // encrypt session
    TPM2B_TYPE(SYM_KEY, (  sizeof(extraKey->t.buffer)
                         + sizeof(session->sessionKey.t.buffer)));
    TPM2B_SYM_KEY        key;               // encryption key
    UINT32               cipherSize = 0;    // size of cipher text

    pAssert(session->sessionKey.t.size + extraKey->t.size <= sizeof(key.t.buffer));

    // Retrieve encrypted data size.
    if(leadingSizeInByte == 2)
    {
        // Extract the first two bytes as the size field as the data size
        // encrypt
        cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
        // advance the buffer
        buffer = &buffer[2];
    }
#ifdef      TPM4B
    else if(leadingSizeInByte == 4)
    {
        // use the first four bytes to indicate the number of bytes to encrypt
        cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
        //advance pointer
        buffer = &buffer[4];
    }
#endif
    else
    {
        pAssert(FALSE);
    }

    // Compute encryption key by concatenating sessionAuth with extra key
    MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
    MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));

    if (session->symmetric.algorithm == TPM_ALG_XOR)

        // XOR parameter encryption formulation:
        //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
        CryptXORObfuscation(session->authHashAlg, &(key.b),
                                   &(session->nonceTPM.b),
                                   nonceCaller, cipherSize, buffer);
    else
        ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
                              session->symmetric.keyBits.aes, &(key.b),
                              nonceCaller, &(session->nonceTPM.b),
                              cipherSize, buffer);
    return;
}
```

11.2.10.9 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Table 147

Error Returns	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```
TPM_RC
CryptParameterDecryption(
    TPM_HANDLE        handle,            // IN: encrypted session handle
    TPM2B            *nonceCaller,       // IN: nonce caller
    UINT32            bufferSize,        // IN: size of parameter buffer
    UINT16            leadingSizeInByte, // IN: the size of the leading size field in
                                         //     byte
    TPM2B_AUTH       *extraKey,          // IN: the authValue
    BYTE             *buffer             // IN/OUT: parameter buffer to be decrypted
    )
{
    SESSION          *session = SessionGet(handle);  // encrypt session
    // The HMAC key is going to be the concatenation of the session key and any
    // additional key material (like the authValue). The size of both of these
    // is the size of the buffer which can contain a TPMT_HA.
    TPM2B_TYPE(HMAC_KEY, (  sizeof(extraKey->t.buffer)
                          + sizeof(session->sessionKey.t.buffer)));
    TPM2B_HMAC_KEY            key;            // decryption key
    UINT32                    cipherSize = 0; // size of cipher text

    pAssert(session->sessionKey.t.size + extraKey->t.size <= sizeof(key.t.buffer));

    // Retrieve encrypted data size.
    if(leadingSizeInByte == 2)
    {
        // The first two bytes of the buffer are the size of the
        // data to be decrypted
        cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
        buffer = &buffer[2];   // advance the buffer
    }
#ifdef  TPM4B
    else if(leadingSizeInByte == 4)
    {
        // the leading size is four bytes so get the four byte size field
        cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
        buffer = &buffer[4];   //advance pointer
    }
#endif
    else
    {
        pAssert(FALSE);
    }
    if(cipherSize > bufferSize)
        return TPM_RC_SIZE;

    // Compute decryption key by concatenating sessionAuth with extra input key
    MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
    MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));

    if(session->symmetric.algorithm == TPM_ALG_XOR)
        // XOR parameter decryption formulation:
        //    XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
        // Call XOR obfuscation function
        CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
                                 &(session->nonceTPM.b), cipherSize, buffer);
    else
        // Assume that it is one of the symmetric block ciphers.
        ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
                            session->symmetric.keyBits.sym,
```



```
                                   &key.b, nonceCaller, &session->nonceTPM.b,
                                   cipherSize, buffer);

    return TPM_RC_SUCCESS;

}
```

11.2.10.10 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```
void
CryptComputeSymmetricUnique(
    TPMI_ALG_HASH    nameAlg,        // IN: object name algorithm
    TPMT_SENSITIVE  *sensitive,      // IN: sensitive area
    TPM2B_DIGEST    *unique          // OUT: unique buffer
    )
{
    HASH_STATE  hashState;

    pAssert(sensitive != NULL && unique != NULL);

    // Compute the public value as the hash of sensitive.symkey || unique.buffer
    unique->t.size = CryptGetHashDigestSize(nameAlg);
    CryptStartHash(nameAlg, &hashState);

    // Add obfuscation value
    CryptUpdateDigest2B(&hashState, &sensitive->seedValue.b);

    // Add sensitive value
    CryptUpdateDigest2B(&hashState, &sensitive->sensitive.any.b);

    CryptCompleteHash2B(&hashState, &unique->b);

    return;
}
#if 0 //%
```

11.2.10.11 CryptComputeSymValue()

This function computes the *seedValue* field in asymmetric sensitive areas.

```
void
CryptComputeSymValue(
    TPM_HANDLE        parentHandle,   // IN: parent handle of the object to be created
    TPMT_PUBLIC      *publicArea,     // IN/OUT: the public area template
    TPMT_SENSITIVE   *sensitive,      // IN: sensitive area
    TPM2B_SEED       *seed,           // IN: the seed
    TPMI_ALG_HASH     hashAlg,        // IN: hash algorithm for KDFa
    TPM2B_NAME       *name            // IN: object name
    )
{
    TPM2B_AUTH   *proof = NULL;

    if(CryptIsAsymAlgorithm(publicArea->type))
    {
        // Generate seedValue only when an asymmetric key is a storage key
        if(publicArea->objectAttributes.decrypt == SET
                && publicArea->objectAttributes.restricted == SET)
        {
            // If this is a primary object in the endorsement hierarchy, use
            // ehProof in the creation of the symmetric seed so that child
            // objects in the endorsement hierarchy are voided on TPM2_Clear()
```

```
            // or TPM2_ChangeEPS()
            if(   parentHandle == TPM_RH_ENDORSEMENT
               && publicArea->objectAttributes.fixedTPM == SET)
                proof = &gp.ehProof;
        }
        else
        {
            sensitive->seedValue.t.size = 0;
            return;
        }
    }

    // For all object types, the size of seedValue is the digest size of nameAlg
    sensitive->seedValue.t.size = CryptGetHashDigestSize(publicArea->nameAlg);

    // Compute seedValue using implementation-dependent method
    _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
                                sensitive->seedValue.t.buffer,
                                hashAlg,
                                &seed->b,
                                "seedValue",
                                &name->b,
                                (TPM2B *)proof);
    return;
}
#endif //%
```

11.2.10.12 CryptCreateObject()

This function creates an object. It:

a) fills in the created key in public and sensitive area;

b) creates a random number in sensitive area for symmetric keys; and

c) compute the unique id in public area for symmetric keys.

Table 148

Error Returns	Meaning
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area for a symmetric key
TPM_RC_RANGE	for an RSA key, the exponent is not supported
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme for a keyed hash object
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key

```
TPM_RC
CryptCreateObject(
    TPM_HANDLE                  parentHandle,      // IN/OUT: indication of the seed
                                                   //     source
    TPMT_PUBLIC                *publicArea,        // IN/OUT: public area
    TPMS_SENSITIVE_CREATE      *sensitiveCreate,   // IN: sensitive creation
    TPMT_SENSITIVE             *sensitive          // OUT: sensitive area
    )
{
    // Next value is a placeholder for a random seed that is used in
    // key creation when the parent is not a primary seed. It has the same
    // size as the primary seed.
```

```

    TPM2B_SEED          localSeed;       // data to seed key creation if this
                                         // is not a primary seed

    TPM2B_SEED         *seed = NULL;
    TPM_RC              result = TPM_RC_SUCCESS;

    TPM2B_NAME          name;
    TPM_ALG_ID          hashAlg = CONTEXT_INTEGRITY_HASH_ALG;
    OBJECT             *parent;
    UINT32              counter;

    // Set the sensitive type for the object
    sensitive->sensitiveType = publicArea->type;
    ObjectComputeName(publicArea, &name);

    // For all objects, copy the initial auth data
    sensitive->authValue = sensitiveCreate->userAuth;

    // If this is a permanent handle assume that it is a hierarchy
    if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
    {
        seed = HierarchyGetPrimarySeed(parentHandle);
    }
    else
    {
        // If not hierarchy handle, get parent
        parent = ObjectGet(parentHandle);
        hashAlg = parent->publicArea.nameAlg;

        // Use random value as seed for non-primary objects
        localSeed.t.size = PRIMARY_SEED_SIZE;
        CryptGenerateRandom(PRIMARY_SEED_SIZE, localSeed.t.buffer);
        seed = &localSeed;
    }

    switch(publicArea->type)
    {
#ifdef TPM_ALG_RSA
        // Create RSA key
    case TPM_ALG_RSA:
        result = CryptGenerateKeyRSA(publicArea, sensitive,
                                     hashAlg, seed, &name, &counter);
        break;
#endif // TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        // Create ECC key
    case TPM_ALG_ECC:
        result = CryptGenerateKeyECC(publicArea, sensitive,
                                         hashAlg, seed, &name, &counter);
        break;
#endif // TPM_ALG_ECC

        // Collect symmetric key information
    case TPM_ALG_SYMCIPHER:
        return CryptGenerateKeySymmetric(publicArea, sensitiveCreate,
                                         sensitive, hashAlg, seed, &name);
        break;
    case TPM_ALG_KEYEDHASH:
        return CryptGenerateKeyedHash(publicArea, sensitiveCreate,
                                      sensitive, hashAlg, seed, &name);
        break;
    default:
        pAssert(0);
        break;
```

```
    }
    if(result == TPM_RC_SUCCESS)
    {
        TPM2B_AUTH          *proof = NULL;

        if(publicArea->objectAttributes.decrypt == SET
                && publicArea->objectAttributes.restricted == SET)
        {
            // If this is a primary object in the endorsement hierarchy, use
            // ehProof in the creation of the symmetric seed so that child
            // objects in the endorsement hierarchy are voided on TPM2_Clear()
            // or TPM2_ChangeEPS()
            if(   parentHandle == TPM_RH_ENDORSEMENT
               && publicArea->objectAttributes.fixedTPM == SET)
                proof = &gp.ehProof;

            // For all object types, the size of seedValue is the digest size
            // of its nameAlg
            sensitive->seedValue.t.size
                = CryptGetHashDigestSize(publicArea->nameAlg);

            // Compute seedValue using implementation-dependent method
            _cpri__GenerateSeededRandom(sensitive->seedValue.t.size,
                                        sensitive->seedValue.t.buffer,
                                        hashAlg,
                                        &seed->b,
                                        "seedValuea",
                                        &name.b,
                                        (TPM2B *)proof);
        }
        else
        {
            sensitive->seedValue.t.size = 0;
        }
    }

    return result;

}
```

11.2.10.13 CryptObjectIsPublicConsistent()

This function checks that the key sizes in the public area are consistent. For an asymmetric key, the size of the public key must match the size indicated by the public->parameters.

Checks for the algorithm types matching the key type are handled by the unmarshaling operation.

Table 149

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
BOOL
CryptObjectIsPublicConsistent(
    TPMT_PUBLIC     *publicArea     // IN: public area
    )
{
    BOOL                    OK = TRUE;
    switch (publicArea->type)
    {
#ifdef TPM_ALG_RSA
```



```
            case TPM_ALG_RSA:
                OK = CryptAreKeySizesConsistent(publicArea);
                break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
            case TPM_ALG_ECC:
                {
                    const ECC_CURVE                 *curveValue;

                    // Check that the public point is on the indicated curve.
                    OK = CryptEccIsPointOnCurve(
                                    publicArea->parameters.eccDetail.curveID,
                                    &publicArea->unique.ecc);
                    if(OK)
                    {
                        curveValue = CryptEccGetCurveDataPointer(
                                             publicArea->parameters.eccDetail.curveID);
                        pAssert(curveValue != NULL);

                        // The input ECC curve must be a supported curve
                        // IF a scheme is defined for the curve, then that scheme must
                        // be used.
                        OK =    (curveValue->sign.scheme == TPM_ALG_NULL
                             || (   publicArea->parameters.eccDetail.scheme.scheme
                                 == curveValue->sign.scheme));
                        OK = OK && CryptAreKeySizesConsistent(publicArea);
                    }
                }
                break;
#endif //TPM_ALG_ECC

            default:
                // Symmetric object common checks
                // There is noting to check with a symmetric key that is public only.
                // Also not sure that there is anything useful to be done with it
                // either.
                break;
        }
        return OK;
}
```

11.2.10.14 CryptObjectPublicPrivateMatch()

This function checks the cryptographic binding between the public and sensitive areas.

Table 150

Error Returns	Meaning
TPM_RC_TYPE	the type of the public and private areas are not the same
TPM_RC_FAILURE	crypto error
TPM_RC_BINDING	the public and private areas are not cryptographically matched.

```
TPM_RC
CryptObjectPublicPrivateMatch(
    OBJECT           *object         // IN: the object to check
    )
{
    TPMT_PUBLIC          *publicArea;
    TPMT_SENSITIVE       *sensitive;
    TPM_RC                result = TPM_RC_SUCCESS;
```

```
    BOOL                 isAsymmetric = FALSE;

    pAssert(object != NULL);
    publicArea = &object->publicArea;
    sensitive = &object->sensitive;
    if(publicArea->type != sensitive->sensitiveType)
        return TPM_RC_TYPE;

    switch(publicArea->type)
    {
#ifdef TPM_ALG_RSA
    case TPM_ALG_RSA:
        isAsymmetric = TRUE;
        // The public and private key sizes need to be consistent
        if(sensitive->sensitive.rsa.t.size != publicArea->unique.rsa.t.size/2)
            result = TPM_RC_BINDING;
        else
        // Load key by computing the private exponent
            result = CryptLoadPrivateRSA(object);
        break;
#endif
#ifdef TPM_ALG_ECC
        // This function is called from ObjectLoad() which has already checked to
        // see that the public point is on the curve so no need to repeat that
        // check.
    case TPM_ALG_ECC:
        isAsymmetric = TRUE;
        if(   publicArea->unique.ecc.x.t.size
                != sensitive->sensitive.ecc.t.size)
            result = TPM_RC_BINDING;
        else if(publicArea->nameAlg != TPM_ALG_NULL)
        {
            TPMS_ECC_POINT          publicToCompare;
            // Compute ECC public key
            CryptEccPointMultiply(&publicToCompare,
                                  publicArea->parameters.eccDetail.curveID,
                                  &sensitive->sensitive.ecc, NULL);
            // Compare ECC public key
            if(   (!Memory2BEqual(&publicArea->unique.ecc.x.b,
                                  &publicToCompare.x.b))
               || (!Memory2BEqual(&publicArea->unique.ecc.y.b,
                                  &publicToCompare.y.b)))
                result = TPM_RC_BINDING;
        }
        break;
#endif
    case TPM_ALG_KEYEDHASH:
        break;
    case TPM_ALG_SYMCIPHER:
        if(   (publicArea->parameters.symDetail.sym.keyBits.sym + 7)/8
           != sensitive->sensitive.sym.t.size)
            result = TPM_RC_BINDING;
        break;
    default:
        // The choice here is an assert or a return of a bad type for the object
        pAssert(0);
        break;
    }

    // For asymmetric keys, the algorithm for validating the linkage between
    // the public and private areas is algorithm dependent. For symmetric keys
    // the linkage is based on hashing the symKey and obfuscation values.
    if(   result == TPM_RC_SUCCESS && !isAsymmetric
       && publicArea->nameAlg != TPM_ALG_NULL)
    {
        TPM2B_DIGEST    uniqueToCompare;
```



```

        // Compute unique for symmetric key
        CryptComputeSymmetricUnique(publicArea->nameAlg, sensitive,
                                    &uniqueToCompare);
        // Compare unique
        if(!Memory2BEqual(&publicArea->unique.sym.b,
                          &uniqueToCompare.b))
            result = TPM_RC_BINDING;
    }
    return result;

}
```

11.2.10.15 CryptGetSignHashAlg()

Get the hash algorithm of signature from a TPMT_SIGNATURE structure. It assumes the signature is not NULL This is a function for easy access

```
TPMI_ALG_HASH
CryptGetSignHashAlg(
    TPMT_SIGNATURE  *auth          // IN: signature
    )
{
    pAssert(auth->sigAlg != TPM_ALG_NULL);

    // Get authHash algorithm based on signing scheme
    switch(auth->sigAlg)
    {

#ifdef  TPM_ALG_RSA
        case TPM_ALG_RSASSA:
            return auth->signature.rsassa.hash;

        case TPM_ALG_RSAPSS:
            return auth->signature.rsapss.hash;

    #endif //TPM_ALG_RSA

    #ifdef TPM_ALG_ECC
        case TPM_ALG_ECDSA:
            return auth->signature.ecdsa.hash;

    #endif //TPM_ALG_ECC

        case TPM_ALG_HMAC:
            return auth->signature.hmac.hashAlg;

        default:
            return TPM_ALG_NULL;
    }
}
```

11.2.10.16 CryptIsSplitSign()

This function us used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```
BOOL
CryptIsSplitSign(
    TPM_ALG_ID      scheme         // IN: the algorithm selector
    )
{
```

```
    if(   scheme != scheme
#   ifdef    TPM_ALG_ECDAA
        || scheme == TPM_ALG_ECDAA
#   endif    // TPM_ALG_ECDAA

      )
        return TRUE;
    return FALSE;
}
```

11.2.10.17 CryptIsSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```
BOOL
CryptIsSignScheme(
    TPMI_ALG_ASYM_SCHEME     scheme
    )
{
    BOOL            isSignScheme = FALSE;

    switch(scheme)
    {
#ifdef TPM_ALG_RSA
        // If RSA is implemented, then both signing schemes are required
    case TPM_ALG_RSASSA:
    case TPM_ALG_RSAPSS:
        isSignScheme = TRUE;
        break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        // If ECC is implemented ECDSA is required
    case TPM_ALG_ECDSA:
#ifdef  TPM_ALG_ECDAA
        // ECDAA is optional
    case TPM_ALG_ECDAA:
#endif
#ifdef   TPM_ALG_ECSCHNORR
        // Schnorr is also optional
    case TPM_ALG_ECSCHNORR:
#endif
#ifdef  TPM_ALG_SM2
    case TPM_ALG_SM2:
#endif
        isSignScheme = TRUE;
        break;
#endif //TPM_ALG_ECC
    default:
        break;
    }
    return isSignScheme;
}
```

11.2.10.18 CryptIsDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
BOOL
CryptIsDecryptScheme(
    TPMI_ALG_ASYM_SCHEME     scheme
    )
{
```



```
    BOOL        isDecryptScheme = FALSE;

    switch(scheme)
    {
#ifdef TPM_ALG_RSA
        // If RSA is implemented, then both decrypt schemes are required
    case TPM_ALG_RSAES:
    case TPM_ALG_OAEP:
         isDecryptScheme = TRUE;
        break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        // If ECC is implemented ECDH is required
    case TPM_ALG_ECDH:
#ifdef TPM_ALG_SM2
    case TPM_ALG_SM2:
#endif
#ifdef  TPM_ALG_ECMQV
    case TPM_ALG_ECMQV:
#endif
        isDecryptScheme = TRUE;
        break;
#endif //TPM_ALG_ECC
    default:
        break;
    }
    return isDecryptScheme;
}
```

11.2.10.19 CryptSelectSignScheme()

This function is used by the attestation and signing commands. It implements the rules for selecting the signature scheme to use in signing. This function requires that the signing key either be TPM_RH_NULL or be loaded.

If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both object and input scheme has a non-NULL scheme algorithm, if the schemes are compatible, the input scheme will be chosen.

Table 151

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*

```
TPM_RC
CryptSelectSignScheme(
    TPMI_DH_OBJECT       signHandle,    // IN: handle of signing key
    TPMT_SIG_SCHEME     *scheme         // IN/OUT: signing scheme
    )
{
    OBJECT              *signObject;
    TPMT_SIG_SCHEME     *objectScheme;
    TPMT_PUBLIC         *publicArea;
    TPM_RC               result = TPM_RC_SUCCESS;

    // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
    // of the setting of scheme
```

```
    if(signHandle == TPM_RH_NULL)
    {
        scheme->scheme = TPM_ALG_NULL;
        scheme->details.any.hashAlg = TPM_ALG_NULL;
    }
    else
    {
        // sign handle is not NULL so...
        // Get sign object pointer
        signObject = ObjectGet(signHandle);
        publicArea = &signObject->publicArea;

        // is this a signing key?
        if(!publicArea->objectAttributes.sign)
            result = TPM_RC_KEY;
        else
        {
            // "parms" defined to avoid long code lines.
            TPMU_PUBLIC_PARMS   *parms = &publicArea->parameters;
            if(CryptIsAsymAlgorithm(publicArea->type))
                objectScheme = (TPMT_SIG_SCHEME *)&parms->asymDetail.scheme;
            else
                objectScheme = (TPMT_SIG_SCHEME *)&parms->keyedHashDetail.scheme;

            // If the object doesn't have a default scheme, then use the
            // input scheme.
            if(objectScheme->scheme == TPM_ALG_NULL)
            {
                // Input and default can't both be NULL
                if(scheme->scheme == TPM_ALG_NULL)
                    result = TPM_RC_SCHEME;

                // Assume that the scheme is compatible with the key. If not,
                // we will generate an error in the signing operation.

            }
            else if(scheme->scheme == TPM_ALG_NULL)
            {
                // input scheme is NULL so use default

                // First, check to see if the default requires that the caller
                // provided scheme data
                if(CryptIsSplitSign(objectScheme->scheme))
                    result = TPM_RC_SCHEME;
                else
                {
                    scheme->scheme = objectScheme->scheme;
                    scheme->details.any.hashAlg
                                = objectScheme->details.any.hashAlg;
                }
            }
            else
            {
                // Both input and object have scheme selectors
                // If the scheme and the hash are not the same then...
                if(   objectScheme->scheme != scheme->scheme
                   || (   objectScheme->details.any.hashAlg
                       != scheme->details.any.hashAlg))
                    result = TPM_RC_SCHEME;
            }
        }

    }
    return result;
}
```


11.2.10.20 CryptSign()

Sign a digest with asymmetric key or HMAC. This function is called by attestation commands and the generic TPM2_Sign() command. This function checks the key scheme and digest size. It does not check if the sign operation is allowed for restricted key. It should be checked before the function is called. The function will assert if the key is not a signing key.

Table 152

Error Returns	Meaning
TPM_RC_SCHEME	*signScheme* is not compatible with the signing key type
TPM_RC_VALUE	*digest* value is greater than the modulus of *signHandle* or size of *hashData* does not match hash algorithm in*signScheme* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
CryptSign(
    TPMI_DH_OBJECT       signHandle,    // IN: The handle of sign key
    TPMT_SIG_SCHEME     *signScheme,    // IN: sign scheme.
    TPM2B_DIGEST        *digest,        // IN: The digest being signed
    TPMT_SIGNATURE      *signature      // OUT: signature
    )
{
    OBJECT              *signKey = ObjectGet(signHandle);
    TPM_RC               result = TPM_RC_SCHEME;

    // check if input handle is a sign key
    pAssert(signKey->publicArea.objectAttributes.sign == SET);

    // Must have the private portion loaded.  This check is made during
    // authorization.
    pAssert(signKey->attributes.publicOnly == CLEAR);

    // Initialize signature scheme
    signature->sigAlg = signScheme->scheme;

    // If the signature algorithm is TPM_ALG_NULL, then we are done
    if(signature->sigAlg == TPM_ALG_NULL)
        return TPM_RC_SUCCESS;

    // All the schemes other than TPM_ALG_NULL have a hash algorithm
    TEST_HASH(signScheme->details.any.hashAlg);

    // Initialize signature hash
    // Note: need to do the check for alg null first because the null scheme
    // doesn't have a hashAlg member.
    signature->signature.any.hashAlg = signScheme->details.any.hashAlg;

    // perform sign operation based on different key type
    switch (signKey->publicArea.type)
    {

#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            result = CryptSignRSA(signKey, signScheme, digest, signature);
            break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            result = CryptSignECC(signKey, signScheme, digest, signature);
```

```
            break;
#endif //TPM_ALG_ECC
        case TPM_ALG_KEYEDHASH:
            result = CryptSignHMAC(signKey, signScheme, digest, signature);
            break;
        default:
            break;
    }

    return result;
}
```

11.2.10.21 CryptVerifySignature()

This function is used to verify a signature. It is called by TPM2_VerifySignature() and TPM2_PolicySigned().

Since this operation only requires use of a public key, no consistency checks are necessary for the key to signature type because a caller can load any public key that they like with any scheme that they like. This routine simply makes sure that the signature is correct, whatever the type.

This function requires that *auth* is not a NULL pointer.

Table 153

Error Returns	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```
TPM_RC
CryptVerifySignature(
    TPMI_DH_OBJECT   keyHandle,      // IN: The handle of sign key
    TPM2B_DIGEST    *digest,         // IN: The digest being validated
    TPMT_SIGNATURE  *signature       // IN: signature
    )
{
    // NOTE: ObjectGet will either return a pointer to a loaded object or
    // will assert. It will never return a non-valid value. This makes it save
    // to initialize 'publicArea' with the return value from ObjectGet() without
    // checking it first.
    OBJECT              *authObject = ObjectGet(keyHandle);
    TPMT_PUBLIC         *publicArea = &authObject->publicArea;
    TPM_RC               result = TPM_RC_SCHEME;

    // The input unmarshaling should prevent any input signature from being
    // a NULL signature, but just in case
    if(signature->sigAlg == TPM_ALG_NULL)
        return TPM_RC_SIGNATURE;

    switch (publicArea->type)
    {

#ifdef TPM_ALG_RSA
    case TPM_ALG_RSA:
        result = CryptRSAVerifySignature(authObject, digest, signature);
        break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
```



```
    case TPM_ALG_ECC:
        result = CryptECCVerifySignature(authObject, digest, signature);
        break;

#endif // TMP_ALG_ECC

    case TPM_ALG_KEYEDHASH:
        if(authObject->attributes.publicOnly)
            result = TPM_RCS_HANDLE;
        else
            result = CryptHMACVerifySignature(authObject, digest, signature);
        break;

    default:
        break;
    }
    return result;

}
```

11.2.11 Math functions

11.2.11.1 CryptDivide()

This function interfaces to the math library for large number divide.

Table 154

Error Returns	Meaning
TPM_RC_SIZE	*quotient* or *remainder* is too small to receive the result

```
TPM_RC
CryptDivide(
    TPM2B           *numerator,      // IN: numerator
    TPM2B           *denominator,    // IN: denominator
    TPM2B           *quotient,       // OUT: quotient = numerator / denominator.
    TPM2B           *remainder       // OUT: numerator mod denominator.
    )
{
    pAssert(   numerator != NULL && denominator!= NULL
            && (quotient != NULL || remainder != NULL)
           );
    // assume denominator is not 0
    pAssert(denominator->size != 0);

    return TranslateCryptErrors(_math__Div(numerator,
                                           denominator,
                                           quotient,
                                           remainder)
                               );
}
```

11.2.11.2 CryptCompare()

This function interfaces to the math library for large number, unsigned compare.

Table 155

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
LIB_EXPORT int
CryptCompare(
    const UINT32     aSize,         // IN: size of a
    const BYTE      *a,             // IN: a buffer
    const UINT32     bSize,         // IN: size of b
    const BYTE      *b              // IN: b buffer
    )
{
    return _math__uComp(aSize, a, bSize, b);
}
```

11.2.11.3 CryptCompareSigned()

This function interfaces to the math library for large number, signed compare.

Table 156

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
int
CryptCompareSigned(
    UINT32           aSize,         // IN: size of a
    BYTE            *a,             // IN: a buffer
    UINT32           bSize,         // IN: size of b
    BYTE            *b              // IN: b buffer
    )
{
    return _math__Comp(aSize, a, bSize, b);
}
```

11.2.11.4 CryptGetTestResult

This function returns the results of a self-test function.

NOTE the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult ISO/IEC 11889-3, clause 11.4, "TPM2_GetTestResult".

```
TPM_RC
CryptGetTestResult(
    TPM2B_MAX_BUFFER    *outData        // OUT: test result data
    )
{
    outData->t.size = 0;
    return TPM_RC_SUCCESS;
}
```


11.2.12 Capability Support

11.2.12.1 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Table 157

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```
#ifdef TPM_ALG_ECC //% 5
TPMI_YES_NO
CryptCapGetECCCurve(
    TPM_ECC_CURVE    curveID,       // IN: the starting ECC curve
    UINT32           maxCount,      // IN: count of returned curves
    TPML_ECC_CURVE  *curveList      // OUT: ECC curve list
    )
{
    TPMI_YES_NO       more = NO;
    UINT16            i;
    UINT32            count = _cpri__EccGetCurveCount();
    TPM_ECC_CURVE     curve;

    // Initialize output property list
    curveList->count = 0;

    // The maximum count of curves we may return is MAX_ECC_CURVES
    if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;

    // Scan the eccCurveValues array
    for(i = 0; i < count; i++)
    {
        curve = _cpri__GetCurveIdByIndex(i);
        // If curveID is less than the starting curveID, skip it
        if(curve < curveID)
            continue;

        if(curveList->count < maxCount)
        {
            // If we have not filled up the return list, add more curves to
            // it
            curveList->eccCurves[curveList->count] = curve;
            curveList->count++;
        }
        else
        {
            // If the return list is full but we still have curves
            // available, report this and stop iterating
            more = YES;
            break;
        }

    }

    return more;

}
```

11.2.12.2 CryptCapGetEccCurveNumber()

This function returns the number of ECC curves supported by the TPM.

```
UINT32
CryptCapGetEccCurveNumber(
    void
    )
{
    // There is an array that holds the curve data. Its size divided by the
    // size of an entry is the number of values in the table.
    return _cpri__EccGetCurveCount();
}
#endif //TPM_ALG_ECC //% 5
```

11.2.12.3 CryptAreKeySizesConsistent()

This function validates that the public key size values are consistent for an asymmetric key.

NOTE This is not a comprehensive test of the public key.

Table 158

Return Value	Meaning
TRUE	sizes are consistent
FALSE	sizes are not consistent

```
BOOL
CryptAreKeySizesConsistent(
    TPMT_PUBLIC     *publicArea     // IN: the public area to check
    )
{
    BOOL            consistent = FALSE;

    switch (publicArea->type)
    {
#ifdef TPM_ALG_RSA
        case TPM_ALG_RSA:
            // The key size in bits is filtered by the unmarshaling
            consistent = (    ((publicArea->parameters.rsaDetail.keyBits+7)/8)
                           == publicArea->unique.rsa.t.size);
            break;
#endif //TPM_ALG_RSA

#ifdef TPM_ALG_ECC
        case TPM_ALG_ECC:
            {
                UINT16          keySizeInBytes;
                TPM_ECC_CURVE   curveId = publicArea->parameters.eccDetail.curveID;

                keySizeInBytes = CryptEccGetKeySizeInBytes(curveId);

                consistent =   keySizeInBytes > 0
                            && publicArea->unique.ecc.x.t.size <= keySizeInBytes
                            && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
            }
            break;
#endif //TPM_ALG_ECC
        default:
            break;
    }
```



```

    return consistent;
}
```

11.2.12.4 CryptAlgSetImplemented()

This function initializes the bit vector with one bit for each implemented algorithm. This function is called from _TPM_Init(). The vector of implemented algorithms should be generated by the ISO/IEC 11889-2 parser so that the *g_implementedAlgorithms* vector can be a const. That's not how it is now.

```
void
CryptAlgsSetImplemented(
    void
    )
{
    AlgorithmGetImplementedVector(&g_implementedAlgorithms);
}
```

11.3 Ticket.c

11.3.1 Introduction

Clause 11.3.3 contains the functions used for ticket computations.

11.3.2 Includes

```
#include "InternalRoutines.h"
```

11.3.3 Functions

11.3.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer.

Table 159

Return Value	Meaning
TRUE	It is safe to produce ticket
FALSE	It is not safe to produce ticket

```
BOOL
TicketIsSafe(
    TPM2B           *buffer
    )
{
    TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;
    BYTE            bufferToCompare[sizeof(valueToCompare)];
    BYTE            *marshalBuffer;

    // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
    // it is not safe to generate a ticket
    if(buffer->size < <K>sizeof(valueToCompare))
        return FALSE;
```

```

    marshalBuffer = bufferToCompare;
    TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);
    if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
        return FALSE;
    else
        return TRUE;
}
```

11.3.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
void
TicketComputeVerified(
    TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
    TPM2B_DIGEST        *digest,         // IN: digest
    TPM2B_NAME          *keyName,        // IN: name of key that signed the values
    TPMT_TK_VERIFIED    *ticket          // OUT: verified ticket
    )
{
    TPM2B_AUTH          *proof;
    HMAC_STATE           hmacState;

    // Fill in ticket fields
    ticket->tag = TPM_ST_VERIFIED;
    ticket->hierarchy = hierarchy;

    // Use the proof value of the hierarchy
    proof = HierarchyGetProof(hierarchy);

    // Start HMAC
    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                             &proof->b, &hmacState);

    // add TPM_ST_VERIFIED
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);

    // add digest
    CryptUpdateDigest2B(&hmacState, &digest->b);

    // add key name
    CryptUpdateDigest2B(&hmacState, &keyName->b);

    // complete HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

    return;
}
```


11.3.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```
void
TicketComputeAuth(
    TPM_ST               type,           // IN: the type of ticket.
    TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
    UINT64               timeout,        // IN: timeout
    TPM2B_DIGEST        *cpHashA,        // IN: input cpHashA
    TPM2B_NONCE         *policyRef,      // IN: input policyRef
    TPM2B_NAME          *entityName,     // IN: name of entity
    TPMT_TK_AUTH        *ticket          // OUT: Created ticket
```



```
    )
{
    TPM2B_AUTH          *proof;
    HMAC_STATE           hmacState;

    // Get proper proof
    proof = HierarchyGetProof(hierarchy);

    // Fill in ticket fields
    ticket->tag = type;
    ticket->hierarchy = hierarchy;

    // Start HMAC
    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                             &proof->b, &hmacState);

    // Adding TPM_ST_AUTH
    CryptUpdateDigestInt(&hmacState, sizeof(UINT16), &ticket->tag);

    // Adding timeout
    CryptUpdateDigestInt(&hmacState, sizeof(UINT64), &timeout);

    // Adding cpHash
    CryptUpdateDigest2B(&hmacState, &cpHashA->b);

    // Adding policyRef
    CryptUpdateDigest2B(&hmacState, &policyRef->b);

    // Adding keyName
    CryptUpdateDigest2B(&hmacState, &entityName->b);

    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

    return;
}
```

11.3.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```
void
TicketComputeHashCheck(
    TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy constant for ticket
    TPM_ALG_ID           hashAlg,        // IN: the hash algorithm used to create
                                         //     'digest'
    TPM2B_DIGEST        *digest,         // IN: input digest
    TPMT_TK_HASHCHECK   *ticket          // OUT: Created ticket
    )
{
    TPM2B_AUTH          *proof;
    HMAC_STATE           hmacState;

    // Get proper proof
    proof = HierarchyGetProof(hierarchy);

    // Fill in ticket fields
    ticket->tag = TPM_ST_HASHCHECK;
    ticket->hierarchy = hierarchy;

    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                             &proof->b, &hmacState);

    // Add TPM_ST_HASHCHECK
```

```
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);

    // Add hash algorithm
    CryptUpdateDigestInt(&hmacState, sizeof(hashAlg), &hashAlg);

    // Add digest
    CryptUpdateDigest2B(&hmacState, &digest->b);

    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

    return;
}
```

11.3.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```
void
TicketComputeCreation(
    TPMI_RH_HIERARCHY    hierarchy,      // IN: hierarchy for ticket
    TPM2B_NAME          *name,           // IN: object name
    TPM2B_DIGEST        *creation,       // IN: creation hash
    TPMT_TK_CREATION    *ticket          // OUT: created ticket
    )
{
    TPM2B_AUTH          *proof;
    HMAC_STATE           hmacState;

    // Get proper proof
    proof = HierarchyGetProof(hierarchy);

    // Fill in ticket fields
    ticket->tag = TPM_ST_CREATION;
    ticket->hierarchy = hierarchy;

    ticket->digest.t.size = CryptStartHMAC2B(CONTEXT_INTEGRITY_HASH_ALG,
                                             &proof->b, &hmacState);

    // Add TPM_ST_CREATION
    CryptUpdateDigestInt(&hmacState, sizeof(TPM_ST), &ticket->tag);

    // Add name
    CryptUpdateDigest2B(&hmacState, &name->b);

    // Add creation hash
    CryptUpdateDigest2B(&hmacState, &creation->b);

    // Compute HMAC
    CryptCompleteHMAC2B(&hmacState, &ticket->digest.b);

    return;
}
```


11.4 CryptSelfTest.c

11.4.1 Introduction

The functions in this file are designed to support self-test of cryptographic functions in the TPM. The TPM allows the user to decide whether to run self-test on a demand basis or to run all the self-tests before proceeding.

The self-tests are controlled by a set of bit vectors. The **g_untestedDecryptionAlgorithms** vector has a bit for each decryption algorithm that needs to be tested and **g_untestedEncryptionAlgorithms** has a bit for each encryption algorithm that needs to be tested. Before an algorithm is used, the appropriate vector is checked (indexed using the algorithm ID). If the bit is 1, then the test function should be called.

```
#include    "Global.h"
#include    "CryptoEngine.h"
#include    "InternalRoutines.h"
#include    "AlgorithmCap_fp.h"
#include    "CryptSelfTest_fp.h"
#include    "AlgorithmTests_fp.h"
```

11.4.2 Functions

11.4.2.1 RunSelfTest()

Local function to run self-test

```
static TPM_RC
CryptRunSelfTests(
    ALGORITHM_VECTOR    *toTest         // IN: the vector of the algorithms to test
    )
{
    TPM_ALG_ID           alg;

    // For each of the algorithms that are in the toTestVecor, need to run a
    // test
    for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
    {
        if(TEST_BIT(alg, *toTest))
        {
            TPM_RC          result = CryptTestAlgorithm(alg, toTest);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
    }
    return TPM_RC_SUCCESS;
}
```

11.4.2.2 CryptSelfTest()

This function is called to start/complete a full self-test. If *fullTest* is NO, then only the untested algorithms will be run. If *fullTest* is YES, then *g_untestedDecryptionAlgorithms* is reinitialized and then all tests are run. This implementation of the reference design does not support processing outside the framework of a TPM command. As a consequence, this command does not complete until all tests are done. Since this can take a long time, the TPM will check after each test to see if the command is canceled. If so, then the TPM will returned TPM_RC_CANCELLED. To continue with the self-tests, call TPM2_SelfTest(*fullTest* == No) and the TPM will complete the testing.

Table 160

Error Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```
LIB_EXPORT
TPM_RC
CryptSelfTest(
    TPMI_YES_NO      fullTest       // IN: if full test is required
```

```
    )
{
    if(g_forceFailureMode)
        FAIL(FATAL_ERROR_FORCED);

    // If the caller requested a full test, then reset the to test vector so that
    // all the tests will be run
    if(fullTest == YES)
    {
        MemoryCopy(g_toTest,
                   g_implementedAlgorithms,
                   sizeof(g_toTest), sizeof(g_toTest));
    }
    return CryptRunSelfTests(&g_toTest);
}
```

11.4.2.3 CryptIncrementalSelfTest()

This function is used to perform an incremental self-test. This implementation will perform the *toTest* values before returning. That is, it assumes that the TPM cannot perform background tasks between commands.

This command may be canceled. If it is, then there is no return result. However, this command can be run again and the incremental progress will not be lost.

Table 161

Error Returns	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if *toTest* list is not empty
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
TPM_RC
CryptIncrementalSelfTest(
    TPML_ALG         *toTest,        // IN: list of algorithms to be tested
    TPML_ALG         *toDoList       // OUT: list of algorithms needing test
    )
{
    ALGORITHM_VECTOR     toTestVector = {0};
    TPM_ALG_ID           alg;
    UINT32                  i;


    pAssert(toTest != NULL && toDoList != NULL);
    if(toTest->count > 0)
    {
        // Transcribe the toTest list into the toTestVector
        for(i = 0; i < toTest->count; i++)
        {
            TPM_ALG_ID      alg = toTest->algorithms[i];

            // make sure that the algorithm value is not out of range
            if((alg > TPM_ALG_LAST) ||  !TEST_BIT(alg, g_implementedAlgorithms))
                return TPM_RC_VALUE;
            SET_BIT(alg, toTestVector);
        }
        // Run the test
        if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
            return TPM_RC_CANCELED;
    }
    // Fill in the toDoList with the algorithms that are still untested
```

```
    toDoList->count = 0;

    for(alg = TPM_ALG_FIRST;
        toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
        alg++)
    {
        if(TEST_BIT(alg, g_toTest))
            toDoList->algorithms[toDoList->count++] = alg;
    }
    return TPM_RC_SUCCESS;
}
```

11.4.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms. This should not be called unless CryptAlgsSetImplemented() has been called

```
void
CryptInitializeToTest(
    void
    )
{
    MemoryCopy(g_toTest,
               g_implementedAlgorithms,
               sizeof(g_toTest),
               sizeof(g_toTest));
    // Setting the algorithm to null causes the test function to just clear
    // out any algorithms for which there is no test.
    CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);

    return;
}
```

11.4.2.5 CryptTestAlgorithm()

Only point of contact with the actual self tests. If a self-test fails, there is no return and the TPM goes into failure mode. The call to TestAlgorithm() uses an algorithms selector and a bit vector. When the test is run, the corresponding bit in *toTest* and in *g_toTest* is CLEAR. If *toTest* is NULL, then only the bit in *g_toTest* is CLEAR. There is a special case for the call to TestAlgorithm(). When *alg* is TPM_ALG_ERROR, TestAlgorithm() will CLEAR any bit in *toTest* for which it has no test. This allows the knowledge about which algorithms have test to be accessed through the interface that provides the test.

Table 162

Error Returns	Meaning
TPM_RC_SUCCESS	test complete
TPM_RC_CANCELED	test was canceled

```
LIB_EXPORT
TPM_RC
CryptTestAlgorithm(
    TPM_ALG_ID          alg,
    ALGORITHM_VECTOR   *toTest
    )
{
    TPM_RC                    result = TPM_RC_SUCCESS;
#ifdef SELF_TEST
    TPM_RC TestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR *toTest);
    result = TestAlgorithm(alg, toTest);
```

```
#else
    // If this is an attempt to determine the algorithms for which there is a
    // self test, pretend that all of them do. We do that by not clearing any
    // of the algorithm bits. When/if this function is called to run tests, it
    // will over report. This can be changed so that any call to check on which
    // algorithms have tests, 'toTest' can be cleared.
    if(alg != TPM_ALG_ERROR)
    {
        CLEAR_BIT(alg, g_toTest);
        if(toTest != NULL)
            CLEAR_BIT(alg, *toTest);
    }
#endif
    return result;
}
```


Annex A
(informative)
Implementation Dependent

A.1 Introduction

This header file contains definitions that are derived from the values in the annexes of ISO/IEC 11889-2. This file would change based on the implementation.

The values shown in this version of the file reflect the example settings in ISO/IEC 11889-2.

A.2 Implementation.h

```
#ifndef _IMPLEMENTATION_H
#define _IMPLEMENTATION_H
    #include    "BaseTypes.h"
#undef TRUE
#undef FALSE
```

Change these definitions to turn all algorithms or commands on or off

```
#define      ALG_YES      YES
#define      ALG_NO       NO
#define      CC_YES       YES
#define      CC_NO        NO
```

From the ISO/IEC 11889-2, Table A.1, “Defines for SHA1 Hash Values”

```
#define   SHA1_DIGEST_SIZE    20
#define   SHA1_BLOCK_SIZE     64
#define   SHA1_DER_SIZE       15
#define   SHA1_DER            {\
    0x30,0x21,0x30,0x09,0x06,0x05,0x2B,0x0E,0x03,0x02,0x1A,0x05,0x00,0x04,0x14}
```

From the ISO/IEC 11889-2, Table A.2, “Defines for SHA256 Hash Values”

```
#define   SHA256_DIGEST_SIZE    32
#define   SHA256_BLOCK_SIZE     64
#define   SHA256_DER_SIZE       19
#define   SHA256_DER            {\
    0x30,0x31,0x30,0x0D,0x06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X01,    \
    0X05,0X00,0X04,0X20}
```

From the ISO/IEC 11889-2, Table A.3, “Defines for SHA384 Hash Values”

```
#define   SHA384_DIGEST_SIZE    48
#define   SHA384_BLOCK_SIZE     128
#define   SHA384_DER_SIZE       19
#define   SHA384_DER            {\
    0x30,0x41,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X02,    \
    0X05,0X00,0X04,0X30}
```

From the ISO/IEC 11889-2, Table A.4, “Defines for SHA512 Hash Values”

```
#define   SHA512_DIGEST_SIZE    64
#define   SHA512_BLOCK_SIZE     128
#define   SHA512_DER_SIZE       19
#define   SHA512_DER            {\
    0x30,0x51,0x30,0x0D,0X06,0X09,0X60,0X86,0X48,0X01,0X65,0X03,0X04,0X02,0X03,    \
    0X05,0X00,0X04,0X40}
```

From the ISO/IEC 11889-2, Table A.5, “SM3_256 Hash Values”

```
#define  SM3_256_DIGEST_SIZE    32
#define  SM3_256_BLOCK_SIZE     64
#define  SM3_256_DER_SIZE       18
#define  SM3_256_DER            {\
    0x30,0x30,0x30,0x0C,0X06,0X08,0X2A,0X81,0X1C,0X81,0X45,0X01,0X83,0X11,0X05,    \
    0X00,0X04,0X20}
```

From the ISO/IEC 11889-2, Table A.6, “Defines for Architectural Limits Values”

```
#define  MAX_SESSION_NUMBER    3
```

From the ISO/IEC 11889-2, Table B.1, “Defines for Logic Values”

```
#define  YES      1
#define  NO       0
#define  TRUE     1
#define  FALSE    0
#define  SET      1
#define  CLEAR    0
```

From the ISO/IEC 11889-2, Table B.2, “Defines for Processor Values”

```
#define  BIG_ENDIAN_TPM       NO
#define  LITTLE_ENDIAN_TPM    YES
#define  NO_AUTO_ALIGN        NO
```

From the ISO/IEC 11889-2, Table B.3, “Defines for Implemented Algorithms Implemented”

```
#define  ALG_RSA                 ALG_YES
#define  ALG_SHA1                ALG_YES
#define  ALG_HMAC                ALG_YES
#define  ALG_AES                 ALG_YES
#define  ALG_MGF1                ALG_YES
#define  ALG_XOR                 ALG_YES
#define  ALG_KEYEDHASH           ALG_YES
#define  ALG_SHA256              ALG_YES
#define  ALG_SHA384              ALG_YES
#define  ALG_SHA512              ALG_NO
#define  ALG_SM3_256             ALG_NO
#define  ALG_SM4                 ALG_NO
#define  ALG_RSASSA              (ALG_YES*ALG_RSA)
#define  ALG_RSAES               (ALG_YES*ALG_RSA)
#define  ALG_RSAPSS              (ALG_YES*ALG_RSA)
#define  ALG_OAEP                (ALG_YES*ALG_RSA)
#define  ALG_ECC                 ALG_YES
#define  ALG_ECDH                (ALG_YES*ALG_ECC)
#define  ALG_ECDSA               (ALG_YES*ALG_ECC)
#define  ALG_ECDAA               (ALG_YES*ALG_ECC)
#define  ALG_SM2                 (ALG_NO*ALG_ECC)
#define  ALG_ECSCHNORR           (ALG_YES*ALG_ECC)
#define  ALG_ECMQV               (ALG_NO*ALG_ECC)
#define  ALG_SYMCIPHER           ALG_YES
#define  ALG_CAMELLIA            ALG_YES
#define  ALG_KDF1_SP800_56a      (ALG_YES*ALG_ECC)
#define  ALG_KDF2                ALG_NO
#define  ALG_KDF1_SP800_108      ALG_YES
#define  ALG_CTR                 ALG_YES
#define  ALG_OFB                 ALG_YES
#define  ALG_CBC                 ALG_YES
#define  ALG_CFB                 ALG_YES
#define  ALG_ECB                 ALG_YES
```

From the ISO/IEC 11889-2, Table B.4, “Defines for Implemented Commands Implemented”

```
#define  CC_ActivateCredential            CC_YES
#define  CC_Certify                       CC_YES
#define  CC_CertifyCreation               CC_YES
#define  CC_ChangeEPS                     CC_YES
#define  CC_ChangePPS                     CC_YES
#define  CC_Clear                         CC_YES
#define  CC_ClearControl                  CC_YES
#define  CC_ClockRateAdjust               CC_YES
#define  CC_ClockSet                      CC_YES
#define  CC_Commit                        ALG_ECC
#define  CC_ContextLoad                   CC_YES
#define  CC_ContextSave                   CC_YES
#define  CC_Create                        CC_YES
#define  CC_CreatePrimary                 CC_YES
#define  CC_DictionaryAttackLockReset     CC_YES
#define  CC_DictionaryAttackParameters    CC_YES
#define  CC_Duplicate                     CC_YES
#define  CC_ECC_Parameters                ALG_ECC
#define  CC_ECDH_KeyGen                   ALG_ECC
#define  CC_ECDH_ZGen                     ALG_ECC
#define  CC_EncryptDecrypt                CC_YES
#define  CC_EventSequenceComplete         CC_YES
#define  CC_EvictControl                  CC_YES
#define  CC_FieldUpgradeData              CC_NO
#define  CC_FieldUpgradeStart             CC_NO
#define  CC_FirmwareRead                  CC_NO
#define  CC_FlushContext                  CC_YES
#define  CC_GetCapability                 CC_YES
#define  CC_GetCommandAuditDigest         CC_YES
#define  CC_GetRandom                     CC_YES
#define  CC_GetSessionAuditDigest         CC_YES
#define  CC_GetTestResult                 CC_YES
#define  CC_GetTime                       CC_YES
#define  CC_Hash                          CC_YES
#define  CC_HashSequenceStart             CC_YES
#define  CC_HierarchyChangeAuth           CC_YES
#define  CC_HierarchyControl              CC_YES
#define  CC_HMAC                          CC_YES
#define  CC_HMAC_Start                    CC_YES
#define  CC_Import                        CC_YES
#define  CC_IncrementalSelfTest           CC_YES
#define  CC_Load                          CC_YES
#define  CC_LoadExternal                  CC_YES
#define  CC_MakeCredential                CC_YES
#define  CC_NV_Certify                    CC_YES
#define  CC_NV_ChangeAuth                 CC_YES
#define  CC_NV_DefineSpace                CC_YES
#define  CC_NV_Extend                     CC_YES
#define  CC_NV_GlobalWriteLock            CC_YES
#define  CC_NV_Increment                  CC_YES
#define  CC_NV_Read                       CC_YES
#define  CC_NV_ReadLock                   CC_YES
#define  CC_NV_ReadPublic                 CC_YES
#define  CC_NV_SetBits                    CC_YES
#define  CC_NV_UndefineSpace              CC_YES
#define  CC_NV_UndefineSpaceSpecial       CC_YES
#define  CC_NV_Write                      CC_YES
#define  CC_NV_WriteLock                  CC_YES
#define  CC_ObjectChangeAuth              CC_YES
#define  CC_PCR_Allocate                  CC_YES
#define  CC_PCR_Event                     CC_YES
#define  CC_PCR_Extend                    CC_YES
#define  CC_PCR_Read                      CC_YES
#define  CC_PCR_Reset                     CC_YES
```

```
#define   CC_PCR_SetAuthPolicy              CC_YES
#define   CC_PCR_SetAuthValue               CC_YES
#define   CC_PolicyAuthorize                CC_YES
#define   CC_PolicyAuthValue                CC_YES
#define   CC_PolicyCommandCode              CC_YES
#define   CC_PolicyCounterTimer             CC_YES
#define   CC_PolicyCpHash                   CC_YES
#define   CC_PolicyDuplicationSelect        CC_YES
#define   CC_PolicyGetDigest                CC_YES
#define   CC_PolicyLocality                 CC_YES
#define   CC_PolicyNameHash                 CC_YES
#define   CC_PolicyNV                       CC_YES
#define   CC_PolicyOR                       CC_YES
#define   CC_PolicyPassword                 CC_YES
#define   CC_PolicyPCR                      CC_YES
#define   CC_PolicyPhysicalPresence         CC_YES
#define   CC_PolicyRestart                  CC_YES
#define   CC_PolicySecret                   CC_YES
#define   CC_PolicySigned                   CC_YES
#define   CC_PolicyTicket                   CC_YES
#define   CC_PP_Commands                    CC_YES
#define   CC_Quote                          CC_YES
#define   CC_ReadClock                      CC_YES
#define   CC_ReadPublic                     CC_YES
#define   CC_Rewrap                         CC_YES
#define   CC_RSA_Decrypt                    ALG_RSA
#define   CC_RSA_Encrypt                    ALG_RSA
#define   CC_SelfTest                       CC_YES
#define   CC_SequenceComplete               CC_YES
#define   CC_SequenceUpdate                 CC_YES
#define   CC_SetAlgorithmSet                CC_YES
#define   CC_SetCommandCodeAuditStatus      CC_YES
#define   CC_SetPrimaryPolicy               CC_YES
#define   CC_Shutdown                       CC_YES
#define   CC_Sign                           CC_YES
#define   CC_StartAuthSession               CC_YES
#define   CC_Startup                        CC_YES
#define   CC_StirRandom                     CC_YES
#define   CC_TestParms                      CC_YES
#define   CC_Unseal                         CC_YES
#define   CC_VerifySignature                CC_YES
#define   CC_ZGen_2Phase                    CC_YES
#define   CC_EC_Ephemeral                   CC_YES
#define   CC_PolicyNvWritten                CC_YES
```

From the ISO/IEC 11889-2, Table B.5, “Defines for RSA Algorithm Constants”

```
#define   RSA_KEY_SIZES_BITS     {1024,2048}
#define   RSA_KEY_SIZE_BITS_1024
#define   RSA_KEY_SIZE_BITS_2048
#define   MAX_RSA_KEY_BITS       2048
#define   MAX_RSA_KEY_BYTES      ((MAX_RSA_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.6, “Defines for ECC Algorithm Constants”

```
#define   ECC_CURVES             {TPM_ECC_NIST_P256,TPM_ECC_BN_P256,TPM_ECC_SM2_P256}
#define   ECC_KEY_SIZES_BITS     {256}
#define   ECC_KEY_SIZE_BITS_256
#define   MAX_ECC_KEY_BITS       256
#define   MAX_ECC_KEY_BYTES      ((MAX_ECC_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.7 “Defines for AES Algorithm Constants”

```
#define   AES_KEY_SIZES_BITS           {128,256}
```



```
#define  AES_KEY_SIZE_BITS_128
#define  AES_KEY_SIZE_BITS_256
#define  MAX_AES_KEY_BITS            256
#define  MAX_AES_BLOCK_SIZE_BYTES    16
#define  MAX_AES_KEY_BYTES           ((MAX_AES_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.8, “Defines for SM4 Algorithm Constants”

```
#define  SM4_KEY_SIZES_BITS          {128}
#define  SM4_KEY_SIZE_BITS_128
#define  MAX_SM4_KEY_BITS            128
#define  MAX_SM4_BLOCK_SIZE_BYTES    16
#define  MAX_SM4_KEY_BYTES           ((MAX_SM4_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.9, “Defines for CAMELLIA Algorithm Constants”

```
#define  CAMELLIA_KEY_SIZES_BITS          {128}
#define  CAMELLIA_KEY_SIZE_BITS_128
#define  MAX_CAMELLIA_KEY_BITS            128
#define  MAX_CAMELLIA_BLOCK_SIZE_BYTES    16
#define  MAX_CAMELLIA_KEY_BYTES           ((MAX_CAMELLIA_KEY_BITS+7)/8)
```

From the ISO/IEC 11889-2, Table B.10, “Defines for Symmetric Algorithm Constants”

```
#define  MAX_SYM_KEY_BITS      MAX_AES_KEY_BITS
#define  MAX_SYM_KEY_BYTES     MAX_AES_KEY_BYTES
#define  MAX_SYM_BLOCK_SIZE    MAX_AES_BLOCK_SIZE_BYTES
```

From the ISO/IEC 11889-2, Table B.11, “Defines for Implementation Values”

```
#define  FIELD_UPGRADE_IMPLEMENTED        NO
#define  BSIZE                            UINT16
#define  BUFFER_ALIGNMENT                 4
#define  IMPLEMENTATION_PCR               24
#define  PLATFORM_PCR                     24
#define  DRTM_PCR                         17
#define  HCRTM_PCR                        0
#define  NUM_LOCALITIES                   5
#define  MAX_HANDLE_NUM                   3
#define  MAX_ACTIVE_SESSIONS              64
#define  CONTEXT_SLOT                     UINT16
#define  CONTEXT_COUNTER                  UINT64
#define  MAX_LOADED_SESSIONS              3
#define  MAX_SESSION_NUM                  3
#define  MAX_LOADED_OBJECTS               3
#define  MIN_EVICT_OBJECTS                2
#define  PCR_SELECT_MIN                   ((PLATFORM_PCR+7)/8)
#define  PCR_SELECT_MAX                   ((IMPLEMENTATION_PCR+7)/8)
#define  NUM_POLICY_PCR_GROUP             1
#define  NUM_AUTHVALUE_PCR_GROUP          1
#define  MAX_CONTEXT_SIZE                 2048
#define  MAX_DIGEST_BUFFER                1024
#define  MAX_NV_INDEX_SIZE                2048
#define  MAX_NV_BUFFER_SIZE               1024
#define  MAX_CAP_BUFFER                   1024
#define  NV_MEMORY_SIZE                   16384
#define  NUM_STATIC_PCR                   16
#define  MAX_ALG_LIST_SIZE                64
#define  TIMER_PRESCALE                   100000
#define  PRIMARY_SEED_SIZE                32
#define  CONTEXT_ENCRYPT_ALG              TPM_ALG_AES
#define  CONTEXT_ENCRYPT_KEY_BITS         MAX_SYM_KEY_BITS
#define  CONTEXT_ENCRYPT_KEY_BYTES        ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)
#define  CONTEXT_INTEGRITY_HASH_ALG       TPM_ALG_SHA256
```

```
#define  CONTEXT_INTEGRITY_HASH_SIZE      SHA256_DIGEST_SIZE
#define  PROOF_SIZE                       CONTEXT_INTEGRITY_HASH_SIZE
#define  NV_CLOCK_UPDATE_INTERVAL         12
#define  NUM_POLICY_PCR                   1
#define  MAX_COMMAND_SIZE                 4096
#define  MAX_RESPONSE_SIZE                4096
#define  ORDERLY_BITS                     8
#define  MAX_ORDERLY_COUNT                ((1<<ORDERLY_BITS)-1)
#define  ALG_ID_FIRST                     TPM_ALG_FIRST
#define  ALG_ID_LAST                      TPM_ALG_LAST
#define  MAX_SYM_DATA                     128
#define  MAX_RNG_ENTROPY_SIZE             64
#define  RAM_INDEX_SPACE                  512
#define  RSA_DEFAULT_PUBLIC_EXPONENT      0x00010001
#define  ENABLE_PCR_NO_INCREMENT          YES
#define  CRT_FORMAT_RSA                   YES
#define  PRIVATE_VENDOR_SPECIFIC_BYTES    \
    ((MAX_RSA_KEY_BYTES/2)*(3+CRT_FORMAT_RSA*2))
```

From the ISO/IEC 11889-2, Table 8, “Definition of (UINT16) TPM_ALG_ID Constants <IN/OUT, S>”

```
typedef  UINT16            TPM_ALG_ID;
#define  TPM_ALG_ERROR                (TPM_ALG_ID)(0x0000)
#define  ALG_ERROR_VALUE              0x0000
#define  TPM_ALG_FIRST                (TPM_ALG_ID)(0x0001)
#define  ALG_FIRST_VALUE              0x0001
#if defined ALG_RSA && ALG_RSA == YES
#define  TPM_ALG_RSA                  (TPM_ALG_ID)(0x0001)
#endif
#define  ALG_RSA_VALUE                0x0001
#define  TPM_ALG_SHA                  (TPM_ALG_ID)(0x0004)
#define  ALG_SHA_VALUE                0x0004
#if defined ALG_SHA1 && ALG_SHA1 == YES
#define  TPM_ALG_SHA1                 (TPM_ALG_ID)(0x0004)
#endif
#define  ALG_SHA1_VALUE               0x0004
#if defined ALG_HMAC && ALG_HMAC == YES
#define  TPM_ALG_HMAC                 (TPM_ALG_ID)(0x0005)
#endif
#define  ALG_HMAC_VALUE               0x0005
#if defined ALG_AES && ALG_AES == YES
#define  TPM_ALG_AES                  (TPM_ALG_ID)(0x0006)
#endif
#define  ALG_AES_VALUE                0x0006
#if defined ALG_MGF1 && ALG_MGF1 == YES
#define  TPM_ALG_MGF1                 (TPM_ALG_ID)(0x0007)
#endif
#define  ALG_MGF1_VALUE               0x0007
#if defined ALG_KEYEDHASH && ALG_KEYEDHASH == YES
#define  TPM_ALG_KEYEDHASH            (TPM_ALG_ID)(0x0008)
#endif
#define  ALG_KEYEDHASH_VALUE          0x0008
#if defined ALG_XOR && ALG_XOR == YES
#define  TPM_ALG_XOR                  (TPM_ALG_ID)(0x000A)
#endif
#define  ALG_XOR_VALUE                0x000A
#if defined ALG_SHA256 && ALG_SHA256 == YES
#define  TPM_ALG_SHA256               (TPM_ALG_ID)(0x000B)
#endif
#define  ALG_SHA256_VALUE             0x000B
#if defined ALG_SHA384 && ALG_SHA384 == YES
#define  TPM_ALG_SHA384               (TPM_ALG_ID)(0x000C)
#endif
#define  ALG_SHA384_VALUE             0x000C
#if defined ALG_SHA512 && ALG_SHA512 == YES
```

```
#define  TPM_ALG_SHA512             (TPM_ALG_ID)(0x000D)
#endif
#define  ALG_SHA512_VALUE           0x000D
#define  TPM_ALG_NULL               (TPM_ALG_ID)(0x0010)
#define  ALG_NULL_VALUE             0x0010
#if defined ALG_SM3_256 && ALG_SM3_256 == YES
#define  TPM_ALG_SM3_256            (TPM_ALG_ID)(0x0012)
#endif
#define  ALG_SM3_256_VALUE          0x0012
#if defined ALG_SM4 && ALG_SM4 == YES
#define  TPM_ALG_SM4                (TPM_ALG_ID)(0x0013)
#endif
#define  ALG_SM4_VALUE              0x0013
#if defined ALG_RSASSA && ALG_RSASSA == YES
#define  TPM_ALG_RSASSA             (TPM_ALG_ID)(0x0014)
#endif
#define  ALG_RSASSA_VALUE           0x0014
#if defined ALG_RSAES && ALG_RSAES == YES
#define  TPM_ALG_RSAES              (TPM_ALG_ID)(0x0015)
#endif
#define  ALG_RSAES_VALUE            0x0015
#if defined ALG_RSAPSS && ALG_RSAPSS == YES
#define  TPM_ALG_RSAPSS             (TPM_ALG_ID)(0x0016)
#endif
#define  ALG_RSAPSS_VALUE           0x0016
#if defined ALG_OAEP && ALG_OAEP == YES
#define  TPM_ALG_OAEP               (TPM_ALG_ID)(0x0017)
#endif
#define  ALG_OAEP_VALUE             0x0017
#if defined ALG_ECDSA && ALG_ECDSA == YES
#define  TPM_ALG_ECDSA              (TPM_ALG_ID)(0x0018)
#endif
#define  ALG_ECDSA_VALUE            0x0018
#if defined ALG_ECDH && ALG_ECDH == YES
#define  TPM_ALG_ECDH               (TPM_ALG_ID)(0x0019)
#endif
#define  ALG_ECDH_VALUE             0x0019
#if defined ALG_ECDAA && ALG_ECDAA == YES
#define  TPM_ALG_ECDAA              (TPM_ALG_ID)(0x001A)
#endif
#define  ALG_ECDAA_VALUE            0x001A
#if defined ALG_SM2 && ALG_SM2 == YES
#define  TPM_ALG_SM2                (TPM_ALG_ID)(0x001B)
#endif
#define  ALG_SM2_VALUE              0x001B
#if defined ALG_ECSCHNORR && ALG_ECSCHNORR == YES
#define  TPM_ALG_ECSCHNORR          (TPM_ALG_ID)(0x001C)
#endif
#define  ALG_ECSCHNORR_VALUE        0x001C
#if defined ALG_ECMQV && ALG_ECMQV == YES
#define  TPM_ALG_ECMQV              (TPM_ALG_ID)(0x001D)
#endif
#define  ALG_ECMQV_VALUE            0x001D
#if defined ALG_KDF1_SP800_56a && ALG_KDF1_SP800_56a == YES
#define  TPM_ALG_KDF1_SP800_56a     (TPM_ALG_ID)(0x0020)
#endif
#define  ALG_KDF1_SP800_56a_VALUE   0x0020
#if defined ALG_KDF2 && ALG_KDF2 == YES
#define  TPM_ALG_KDF2               (TPM_ALG_ID)(0x0021)
#endif
#define  ALG_KDF2_VALUE             0x0021
#if defined ALG_KDF1_SP800_108 && ALG_KDF1_SP800_108 == YES
#define  TPM_ALG_KDF1_SP800_108     (TPM_ALG_ID)(0x0022)
#endif
#define  ALG_KDF1_SP800_108_VALUE   0x0022
#if defined ALG_ECC && ALG_ECC == YES
```

```
#define   TPM_ALG_ECC                 (TPM_ALG_ID)(0x0023)
#endif
#define   ALG_ECC_VALUE               0x0023
#if defined ALG_SYMCIPHER && ALG_SYMCIPHER == YES
#define   TPM_ALG_SYMCIPHER           (TPM_ALG_ID)(0x0025)
#endif
#define   ALG_SYMCIPHER_VALUE         0x0025
#if defined ALG_CAMELLIA && ALG_CAMELLIA == YES
#define   TPM_ALG_CAMELLIA            (TPM_ALG_ID)(0x0026)
#endif
#define   ALG_CAMELLIA_VALUE          0x0026
#if defined ALG_CTR && ALG_CTR == YES
#define   TPM_ALG_CTR                 (TPM_ALG_ID)(0x0040)
#endif
#define   ALG_CTR_VALUE               0x0040
#if defined ALG_OFB && ALG_OFB == YES
#define   TPM_ALG_OFB                 (TPM_ALG_ID)(0x0041)
#endif
#define   ALG_OFB_VALUE               0x0041
#if defined ALG_CBC && ALG_CBC == YES
#define   TPM_ALG_CBC                 (TPM_ALG_ID)(0x0042)
#endif
#define   ALG_CBC_VALUE               0x0042
#if defined ALG_CFB && ALG_CFB == YES
#define   TPM_ALG_CFB                 (TPM_ALG_ID)(0x0043)
#endif
#define   ALG_CFB_VALUE               0x0043
#if defined ALG_ECB && ALG_ECB == YES
#define   TPM_ALG_ECB                 (TPM_ALG_ID)(0x0044)
#endif
#define   ALG_ECB_VALUE               0x0044
#define   TPM_ALG_LAST                (TPM_ALG_ID)(0x0044)
#define   ALG_LAST_VALUE              0x0044
```

From the ISO/IEC 11889-2, Table 9, “Definition of (UINT16) {ECC} TPM_ECC_CURVE Constants <IN/OUT, S>“

```
typedef   UINT16             TPM_ECC_CURVE;
#define   TPM_ECC_NONE         (TPM_ECC_CURVE)(0x0000)
#define   TPM_ECC_NIST_P192    (TPM_ECC_CURVE)(0x0001)
#define   TPM_ECC_NIST_P224    (TPM_ECC_CURVE)(0x0002)
#define   TPM_ECC_NIST_P256    (TPM_ECC_CURVE)(0x0003)
#define   TPM_ECC_NIST_P384    (TPM_ECC_CURVE)(0x0004)
#define   TPM_ECC_NIST_P521    (TPM_ECC_CURVE)(0x0005)
#define   TPM_ECC_BN_P256      (TPM_ECC_CURVE)(0x0010)
#define   TPM_ECC_BN_P638      (TPM_ECC_CURVE)(0x0011)
#define   TPM_ECC_SM2_P256     (TPM_ECC_CURVE)(0x0020)
```

From the ISO/IEC 11889-2, Table 12, “Definition of (UINT32) TPM_CC Constants (Numeric Order) <IN/OUT, S>”

```
typedef   UINT32             TPM_CC;
#define   TPM_CC_FIRST                          (TPM_CC)(0x0000011F)
#define   TPM_CC_PP_FIRST                       (TPM_CC)(0x0000011F)
#if defined CC_NV_UndefineSpaceSpecial && CC_NV_UndefineSpaceSpecial == YES
#define   TPM_CC_NV_UndefineSpaceSpecial        (TPM_CC)(0x0000011F)
#endif
#if defined CC_EvictControl && CC_EvictControl == YES
#define   TPM_CC_EvictControl                   (TPM_CC)(0x00000120)
#endif
#if defined CC_HierarchyControl && CC_HierarchyControl == YES
#define   TPM_CC_HierarchyControl               (TPM_CC)(0x00000121)
#endif
#if defined CC_NV_UndefineSpace && CC_NV_UndefineSpace == YES
```

```
#define   TPM_CC_NV_UndefineSpace                    (TPM_CC)(0x00000122)
#endif
#if defined CC_ChangeEPS && CC_ChangeEPS == YES
#define   TPM_CC_ChangeEPS                           (TPM_CC)(0x00000124)
#endif
#if defined CC_ChangePPS && CC_ChangePPS == YES
#define   TPM_CC_ChangePPS                           (TPM_CC)(0x00000125)
#endif
#if defined CC_Clear && CC_Clear == YES
#define   TPM_CC_Clear                               (TPM_CC)(0x00000126)
#endif
#if defined CC_ClearControl && CC_ClearControl == YES
#define   TPM_CC_ClearControl                        (TPM_CC)(0x00000127)
#endif
#if defined CC_ClockSet && CC_ClockSet == YES
#define   TPM_CC_ClockSet                            (TPM_CC)(0x00000128)
#endif
#if defined CC_HierarchyChangeAuth && CC_HierarchyChangeAuth == YES
#define   TPM_CC_HierarchyChangeAuth                 (TPM_CC)(0x00000129)
#endif
#if defined CC_NV_DefineSpace && CC_NV_DefineSpace == YES
#define   TPM_CC_NV_DefineSpace                      (TPM_CC)(0x0000012A)
#endif
#if defined CC_PCR_Allocate && CC_PCR_Allocate == YES
#define   TPM_CC_PCR_Allocate                        (TPM_CC)(0x0000012B)
#endif
#if defined CC_PCR_SetAuthPolicy && CC_PCR_SetAuthPolicy == YES
#define   TPM_CC_PCR_SetAuthPolicy                   (TPM_CC)(0x0000012C)
#endif
#if defined CC_PP_Commands && CC_PP_Commands == YES
#define   TPM_CC_PP_Commands                         (TPM_CC)(0x0000012D)
#endif
#if defined CC_SetPrimaryPolicy && CC_SetPrimaryPolicy == YES
#define   TPM_CC_SetPrimaryPolicy                    (TPM_CC)(0x0000012E)
#endif
#if defined CC_FieldUpgradeStart && CC_FieldUpgradeStart == YES
#define   TPM_CC_FieldUpgradeStart                   (TPM_CC)(0x0000012F)
#endif
#if defined CC_ClockRateAdjust && CC_ClockRateAdjust == YES
#define   TPM_CC_ClockRateAdjust                     (TPM_CC)(0x00000130)
#endif
#if defined CC_CreatePrimary && CC_CreatePrimary == YES
#define   TPM_CC_CreatePrimary                       (TPM_CC)(0x00000131)
#endif
#if defined CC_NV_GlobalWriteLock && CC_NV_GlobalWriteLock == YES
#define   TPM_CC_NV_GlobalWriteLock                  (TPM_CC)(0x00000132)
#endif
#define   TPM_CC_PP_LAST                             (TPM_CC)(0x00000132)
#if defined CC_GetCommandAuditDigest && CC_GetCommandAuditDigest == YES
#define   TPM_CC_GetCommandAuditDigest               (TPM_CC)(0x00000133)
#endif
#if defined CC_NV_Increment && CC_NV_Increment == YES
#define   TPM_CC_NV_Increment                        (TPM_CC)(0x00000134)
#endif
#if defined CC_NV_SetBits && CC_NV_SetBits == YES
#define   TPM_CC_NV_SetBits                          (TPM_CC)(0x00000135)
#endif
#if defined CC_NV_Extend && CC_NV_Extend == YES
#define   TPM_CC_NV_Extend                           (TPM_CC)(0x00000136)
#endif
#if defined CC_NV_Write && CC_NV_Write == YES
#define   TPM_CC_NV_Write                            (TPM_CC)(0x00000137)
#endif
#if defined CC_NV_WriteLock && CC_NV_WriteLock == YES
#define   TPM_CC_NV_WriteLock                        (TPM_CC)(0x00000138)
#endif
```

```
#if defined CC_DictionaryAttackLockReset && CC_DictionaryAttackLockReset == YES
#define   TPM_CC_DictionaryAttackLockReset     (TPM_CC)(0x00000139)
#endif
#if defined CC_DictionaryAttackParameters && CC_DictionaryAttackParameters == YES
#define   TPM_CC_DictionaryAttackParameters    (TPM_CC)(0x0000013A)
#endif
#if defined CC_NV_ChangeAuth && CC_NV_ChangeAuth == YES
#define   TPM_CC_NV_ChangeAuth                 (TPM_CC)(0x0000013B)
#endif
#if defined CC_PCR_Event && CC_PCR_Event == YES
#define   TPM_CC_PCR_Event                     (TPM_CC)(0x0000013C)
#endif
#if defined CC_PCR_Reset && CC_PCR_Reset == YES
#define   TPM_CC_PCR_Reset                     (TPM_CC)(0x0000013D)
#endif
#if defined CC_SequenceComplete && CC_SequenceComplete == YES
#define   TPM_CC_SequenceComplete              (TPM_CC)(0x0000013E)
#endif
#if defined CC_SetAlgorithmSet && CC_SetAlgorithmSet == YES
#define   TPM_CC_SetAlgorithmSet               (TPM_CC)(0x0000013F)
#endif
#if defined CC_SetCommandCodeAuditStatus && CC_SetCommandCodeAuditStatus == YES
#define   TPM_CC_SetCommandCodeAuditStatus     (TPM_CC)(0x00000140)
#endif
#if defined CC_FieldUpgradeData && CC_FieldUpgradeData == YES
#define   TPM_CC_FieldUpgradeData              (TPM_CC)(0x00000141)
#endif
#if defined CC_IncrementalSelfTest && CC_IncrementalSelfTest == YES
#define   TPM_CC_IncrementalSelfTest           (TPM_CC)(0x00000142)
#endif
#if defined CC_SelfTest && CC_SelfTest == YES
#define   TPM_CC_SelfTest                      (TPM_CC)(0x00000143)
#endif
#if defined CC_Startup && CC_Startup == YES
#define   TPM_CC_Startup                       (TPM_CC)(0x00000144)
#endif
#if defined CC_Shutdown && CC_Shutdown == YES
#define   TPM_CC_Shutdown                      (TPM_CC)(0x00000145)
#endif
#if defined CC_StirRandom && CC_StirRandom == YES
#define   TPM_CC_StirRandom                    (TPM_CC)(0x00000146)
#endif
#if defined CC_ActivateCredential && CC_ActivateCredential == YES
#define   TPM_CC_ActivateCredential            (TPM_CC)(0x00000147)
#endif
#if defined CC_Certify && CC_Certify == YES
#define   TPM_CC_Certify                       (TPM_CC)(0x00000148)
#endif
#if defined CC_PolicyNV && CC_PolicyNV == YES
#define   TPM_CC_PolicyNV                      (TPM_CC)(0x00000149)
#endif
#if defined CC_CertifyCreation && CC_CertifyCreation == YES
#define   TPM_CC_CertifyCreation               (TPM_CC)(0x0000014A)
#endif
#if defined CC_Duplicate && CC_Duplicate == YES
#define   TPM_CC_Duplicate                     (TPM_CC)(0x0000014B)
#endif
#if defined CC_GetTime && CC_GetTime == YES
#define   TPM_CC_GetTime                       (TPM_CC)(0x0000014C)
#endif
#if defined CC_GetSessionAuditDigest && CC_GetSessionAuditDigest == YES
#define   TPM_CC_GetSessionAuditDigest         (TPM_CC)(0x0000014D)
#endif
#if defined CC_NV_Read && CC_NV_Read == YES
#define   TPM_CC_NV_Read                       (TPM_CC)(0x0000014E)
#endif
```



```
#if defined CC_NV_ReadLock && CC_NV_ReadLock == YES
#define   TPM_CC_NV_ReadLock                      (TPM_CC)(0x0000014F)
#endif
#if defined CC_ObjectChangeAuth && CC_ObjectChangeAuth == YES
#define   TPM_CC_ObjectChangeAuth                 (TPM_CC)(0x00000150)
#endif
#if defined CC_PolicySecret && CC_PolicySecret == YES
#define   TPM_CC_PolicySecret                     (TPM_CC)(0x00000151)
#endif
#if defined CC_Rewrap && CC_Rewrap == YES
#define   TPM_CC_Rewrap                           (TPM_CC)(0x00000152)
#endif
#if defined CC_Create && CC_Create == YES
#define   TPM_CC_Create                           (TPM_CC)(0x00000153)
#endif
#if defined CC_ECDH_ZGen && CC_ECDH_ZGen == YES
#define   TPM_CC_ECDH_ZGen                        (TPM_CC)(0x00000154)
#endif
#if defined CC_HMAC && CC_HMAC == YES
#define   TPM_CC_HMAC                             (TPM_CC)(0x00000155)
#endif
#if defined CC_Import && CC_Import == YES
#define   TPM_CC_Import                           (TPM_CC)(0x00000156)
#endif
#if defined CC_Load && CC_Load == YES
#define   TPM_CC_Load                             (TPM_CC)(0x00000157)
#endif
#if defined CC_Quote && CC_Quote == YES
#define   TPM_CC_Quote                            (TPM_CC)(0x00000158)
#endif
#if defined CC_RSA_Decrypt && CC_RSA_Decrypt == YES
#define   TPM_CC_RSA_Decrypt                      (TPM_CC)(0x00000159)
#endif
#if defined CC_HMAC_Start && CC_HMAC_Start == YES
#define   TPM_CC_HMAC_Start                       (TPM_CC)(0x0000015B)
#endif
#if defined CC_SequenceUpdate && CC_SequenceUpdate == YES
#define   TPM_CC_SequenceUpdate                   (TPM_CC)(0x0000015C)
#endif
#if defined CC_Sign && CC_Sign == YES
#define   TPM_CC_Sign                             (TPM_CC)(0x0000015D)
#endif
#if defined CC_Unseal && CC_Unseal == YES
#define   TPM_CC_Unseal                           (TPM_CC)(0x0000015E)
#endif
#if defined CC_PolicySigned && CC_PolicySigned == YES
#define   TPM_CC_PolicySigned                     (TPM_CC)(0x00000160)
#endif
#if defined CC_ContextLoad && CC_ContextLoad == YES
#define   TPM_CC_ContextLoad                      (TPM_CC)(0x00000161)
#endif
#if defined CC_ContextSave && CC_ContextSave == YES
#define   TPM_CC_ContextSave                      (TPM_CC)(0x00000162)
#endif
#if defined CC_ECDH_KeyGen && CC_ECDH_KeyGen == YES
#define   TPM_CC_ECDH_KeyGen                      (TPM_CC)(0x00000163)
#endif
#if defined CC_EncryptDecrypt && CC_EncryptDecrypt == YES
#define   TPM_CC_EncryptDecrypt                   (TPM_CC)(0x00000164)
#endif
#if defined CC_FlushContext && CC_FlushContext == YES
#define   TPM_CC_FlushContext                     (TPM_CC)(0x00000165)
#endif
#if defined CC_LoadExternal && CC_LoadExternal == YES
#define   TPM_CC_LoadExternal                     (TPM_CC)(0x00000167)
#endif
```

```
#if defined CC_MakeCredential && CC_MakeCredential == YES
#define  TPM_CC_MakeCredential                (TPM_CC)(0x00000168)
#endif
#if defined CC_NV_ReadPublic && CC_NV_ReadPublic == YES
#define  TPM_CC_NV_ReadPublic                 (TPM_CC)(0x00000169)
#endif
#if defined CC_PolicyAuthorize && CC_PolicyAuthorize == YES
#define  TPM_CC_PolicyAuthorize               (TPM_CC)(0x0000016A)
#endif
#if defined CC_PolicyAuthValue && CC_PolicyAuthValue == YES
#define  TPM_CC_PolicyAuthValue               (TPM_CC)(0x0000016B)
#endif
#if defined CC_PolicyCommandCode && CC_PolicyCommandCode == YES
#define  TPM_CC_PolicyCommandCode             (TPM_CC)(0x0000016C)
#endif
#if defined CC_PolicyCounterTimer && CC_PolicyCounterTimer == YES
#define  TPM_CC_PolicyCounterTimer            (TPM_CC)(0x0000016D)
#endif
#if defined CC_PolicyCpHash && CC_PolicyCpHash == YES
#define  TPM_CC_PolicyCpHash                  (TPM_CC)(0x0000016E)
#endif
#if defined CC_PolicyLocality && CC_PolicyLocality == YES
#define  TPM_CC_PolicyLocality                (TPM_CC)(0x0000016F)
#endif
#if defined CC_PolicyNameHash && CC_PolicyNameHash == YES
#define  TPM_CC_PolicyNameHash                (TPM_CC)(0x00000170)
#endif
#if defined CC_PolicyOR && CC_PolicyOR == YES
#define  TPM_CC_PolicyOR                      (TPM_CC)(0x00000171)
#endif
#if defined CC_PolicyTicket && CC_PolicyTicket == YES
#define  TPM_CC_PolicyTicket                  (TPM_CC)(0x00000172)
#endif
#if defined CC_ReadPublic && CC_ReadPublic == YES
#define  TPM_CC_ReadPublic                    (TPM_CC)(0x00000173)
#endif
#if defined CC_RSA_Encrypt && CC_RSA_Encrypt == YES
#define  TPM_CC_RSA_Encrypt                   (TPM_CC)(0x00000174)
#endif
#if defined CC_StartAuthSession && CC_StartAuthSession == YES
#define  TPM_CC_StartAuthSession              (TPM_CC)(0x00000176)
#endif
#if defined CC_VerifySignature && CC_VerifySignature == YES
#define  TPM_CC_VerifySignature               (TPM_CC)(0x00000177)
#endif
#if defined CC_ECC_Parameters && CC_ECC_Parameters == YES
#define  TPM_CC_ECC_Parameters                (TPM_CC)(0x00000178)
#endif
#if defined CC_FirmwareRead && CC_FirmwareRead == YES
#define  TPM_CC_FirmwareRead                  (TPM_CC)(0x00000179)
#endif
#if defined CC_GetCapability && CC_GetCapability == YES
#define  TPM_CC_GetCapability                 (TPM_CC)(0x0000017A)
#endif
#if defined CC_GetRandom && CC_GetRandom == YES
#define  TPM_CC_GetRandom                     (TPM_CC)(0x0000017B)
#endif
#if defined CC_GetTestResult && CC_GetTestResult == YES
#define  TPM_CC_GetTestResult                 (TPM_CC)(0x0000017C)
#endif
#if defined CC_Hash && CC_Hash == YES
#define  TPM_CC_Hash                          (TPM_CC)(0x0000017D)
#endif
#if defined CC_PCR_Read && CC_PCR_Read == YES
#define  TPM_CC_PCR_Read                      (TPM_CC)(0x0000017E)
#endif
```



```
#if defined CC_PolicyPCR && CC_PolicyPCR == YES
#define  TPM_CC_PolicyPCR                     (TPM_CC)(0x0000017F)
#endif
#if defined CC_PolicyRestart && CC_PolicyRestart == YES
#define  TPM_CC_PolicyRestart                 (TPM_CC)(0x00000180)
#endif
#if defined CC_ReadClock && CC_ReadClock == YES
#define  TPM_CC_ReadClock                     (TPM_CC)(0x00000181)
#endif
#if defined CC_PCR_Extend && CC_PCR_Extend == YES
#define  TPM_CC_PCR_Extend                    (TPM_CC)(0x00000182)
#endif
#if defined CC_PCR_SetAuthValue && CC_PCR_SetAuthValue == YES
#define  TPM_CC_PCR_SetAuthValue              (TPM_CC)(0x00000183)
#endif
#if defined CC_NV_Certify && CC_NV_Certify == YES
#define  TPM_CC_NV_Certify                    (TPM_CC)(0x00000184)
#endif
#if defined CC_EventSequenceComplete && CC_EventSequenceComplete == YES
#define  TPM_CC_EventSequenceComplete         (TPM_CC)(0x00000185)
#endif
#if defined CC_HashSequenceStart && CC_HashSequenceStart == YES
#define  TPM_CC_HashSequenceStart             (TPM_CC)(0x00000186)
#endif
#if defined CC_PolicyPhysicalPresence && CC_PolicyPhysicalPresence == YES
#define  TPM_CC_PolicyPhysicalPresence        (TPM_CC)(0x00000187)
#endif
#if defined CC_PolicyDuplicationSelect && CC_PolicyDuplicationSelect == YES
#define  TPM_CC_PolicyDuplicationSelect       (TPM_CC)(0x00000188)
#endif
#if defined CC_PolicyGetDigest && CC_PolicyGetDigest == YES
#define  TPM_CC_PolicyGetDigest               (TPM_CC)(0x00000189)
#endif
#if defined CC_TestParms && CC_TestParms == YES
#define  TPM_CC_TestParms                     (TPM_CC)(0x0000018A)
#endif
#if defined CC_Commit && CC_Commit == YES
#define  TPM_CC_Commit                        (TPM_CC)(0x0000018B)
#endif
#if defined CC_PolicyPassword && CC_PolicyPassword == YES
#define  TPM_CC_PolicyPassword                (TPM_CC)(0x0000018C)
#endif
#if defined CC_ZGen_2Phase && CC_ZGen_2Phase == YES
#define  TPM_CC_ZGen_2Phase                   (TPM_CC)(0x0000018D)
#endif
#if defined CC_EC_Ephemeral && CC_EC_Ephemeral == YES
#define  TPM_CC_EC_Ephemeral                  (TPM_CC)(0x0000018E)
#endif
#if defined CC_PolicyNvWritten && CC_PolicyNvWritten == YES
#define  TPM_CC_PolicyNvWritten               (TPM_CC)(0x0000018F)
#endif
#define  TPM_CC_LAST                          (TPM_CC)(0x0000018F)
#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif
#define MAX_HASH_BLOCK_SIZE  (                   \
    MAX(ALG_SHA1 * SHA1_BLOCK_SIZE,              \
    MAX(ALG_SHA256 * SHA256_BLOCK_SIZE,          \
    MAX(ALG_SHA384 * SHA384_BLOCK_SIZE,          \
    MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE,        \
    MAX(ALG_SHA512 * SHA512_BLOCK_SIZE,          \
    0 ))))))
#define MAX_DIGEST_SIZE      (                   \
    MAX(ALG_SHA1 * SHA1_DIGEST_SIZE,             \
    MAX(ALG_SHA256 * SHA256_DIGEST_SIZE,         \
    MAX(ALG_SHA384 * SHA384_DIGEST_SIZE,         \
```

```
    MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE,       \
    MAX(ALG_SHA512 * SHA512_DIGEST_SIZE,         \
    0 ))))))
#if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
#error "Hash data not valid"
#endif
#define HASH_COUNT (ALG_SHA1+ALG_SHA256+ALG_SHA384+ALG_SM3_256+ALG_SHA512)
#endif  // _IMPLEMENTATION_H
```

Annex B
(informative)
Cryptographic Library Interface

B.1 Introduction

The files in Annex B provide cryptographic support functions for the TPM.

When possible, the functions in these files make calls to functions that are provided by a cryptographic library (for Annex B , it is OpenSSL). In many cases, there is a mismatch between the function performed by the cryptographic library and the function needed by the TPM. In those cases, a function is provided in the code in Annex B .

There are cases where the cryptographic library could have been used for a specific function but not all functions of the same group.

EXAMPLE 1 The OpenSSL version of CFB was not suitable for the requirements of the TPM. Rather than have one symmetric mode be provided in this code with the remaining modes provided by OpenSSL, all the symmetric modes are provided in this code.

The provided cryptographic code is believed to be functionally correct but it might not be conformant with all applicable standards. Still, the implementation meets the major objective of the implementation, which is to demonstrate proper TPM behavior. It is not an objective of this implementation to be submitted for certification.

EXAMPLE 2 The RSA key generation schemes produces serviceable RSA keys but the method is not compliant with FIPS 186-3.

B.2 Integer Format

The big integers passed to/from the function interfaces in the crypto engine are in BYTE buffers that have the same format used in ISO/IEC 11889-1 that states:

"An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

B.3 CryptoEngine.h

B.3.1. Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
#ifndef _CRYPT_PRI_H
#define _CRYPT_PRI_H
#include    <stddef.h>
#include    "TpmBuildSwitches.h"
#include    "BaseTypes.h"
#include    "TpmError.h"
#include    "swap.h"
#include    "Implementation.h"
#include    "TPMB.h"
#include    "bool.h"
#include    "Platform.h"
#ifndef NULL
#define NULL    0
#endif
typedef UINT16  NUMBYTES;       // When a size is a number of bytes
typedef UINT32  NUMDIGITS;      // When a size is a number of "digits"
```

B.3.2. General Purpose Macros

```
#ifndef MAX
#   define MAX(a, b) ((a) > (b) ? (a) : b)
#endif
```

This is the definition of a bit array with one bit per algorithm

```
typedef BYTE    ALGORITHM_VECTOR[(ALG_LAST_VALUE + 7) / 8];
```

B.3.3. Self-test

This structure is used to contain self-test tracking information for the crypto engine. Each of the major modules is given a 32-bit value in which it may maintain its own self test information. The convention for this state is that when all of the bits in this structure are 0, all functions need to be tested.

```
typedef struct {
    UINT32      rng;
    UINT32      hash;
    UINT32      sym;
#ifdef TPM_ALG_RSA
    UINT32      rsa;
#endif
#ifdef  TPM_ALG_ECC
    UINT32      ecc;
#endif
} CRYPTO_SELF_TEST_STATE;
```

B.3.4. Hash-related Structures

```
typedef struct {
    const TPM_ALG_ID      alg;
    const NUMBYTES        digestSize;
    const NUMBYTES        blockSize;
    const NUMBYTES        derSize;
    const BYTE            der[20];
} HASH_INFO;
```

This value will change with each implementation. The value of 16 is used to account for any slop in the context values. The overall size needs to be as large as any of the hash contexts. The structure needs to start on an alignment boundary and be an even multiple of the alignment

```
#define ALIGNED_SIZE(x, b) ((((x) + (b) - 1) / (b)) * (b))
#define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
#define MAX_HASH_STATE_SIZE_ALIGNED                                           \
                    ALIGNED_SIZE(MAX_HASH_STATE_SIZE, CRYPTO_ALIGNMENT)
```

This is an byte array that will hold any of the hash contexts.

```
typedef CRYPTO_ALIGNED BYTE ALIGNED_HASH_STATE[MAX_HASH_STATE_SIZE_ALIGNED];
```

Macro to align an address to the next higher size

```
#define AlignPointer(address, align)                                          \
   ((((intptr_t)&(address)) + (align - 1)) & ~(align - 1))
```

Macro to test alignment

```
#define IsAddressAligned(address, align)                                      \
                    (((intptr_t)(address) & (align - 1)) == 0)
```


This is the structure that is used for passing a context into the hashing functions. It should be the same size as the function context used within the hashing functions. This is checked when the hash function is initialized. This version uses a new layout for the contexts and a different definition. The state buffer is an array of HASH_UNIT values so that a decent compiler will put the structure on a HASH_UNIT boundary. If the structure is not properly aligned, the code that manipulates the structure will copy to a properly aligned structure before it is used and copy the result back. This just makes things slower.

```
typedef struct _HASH_STATE
{
    ALIGNED_HASH_STATE       state;
    TPM_ALG_ID               hashAlg;
} CPRI_HASH_STATE, *PCPRI_HASH_STATE;
extern const HASH_INFO   g_hashData[HASH_COUNT + 1];
```

This is for the external hash state. This implementation assumes that the size of the exported hash state is no larger than the internal hash state. There is a compile-time check to make sure that this is true.

```
typedef struct {
    ALIGNED_HASH_STATE      buffer;
    TPM_ALG_ID              hashAlg;
} EXPORT_HASH_STATE;
typedef enum {
    IMPORT_STATE,       // Converts externally formatted state to internal
    EXPORT_STATE        // Converts internal formatted state to external
} IMPORT_EXPORT;
```

Values and structures for the random number generator. These values are defined in this header file so that the size of the RNG state can be known to TPM. lib. This allows the allocation of some space in NV memory for the state to be stored on an orderly shutdown. The GET_PUT enum is used by _cpri__DrbgGetPutState() to indicate the direction of data flow.

```
typedef enum {
    GET_STATE,      // Get the state to save to NV
    PUT_STATE       // Restore the state from NV
} GET_PUT;
```

The DRBG based on a symmetric block cipher is defined by three values,

a) the key size

b) the block size (the IV size)

c) the symmetric algorithm

```
#define DRBG_KEY_SIZE_BITS      MAX_AES_KEY_BITS
#define DRBG_IV_SIZE_BITS       (MAX_AES_BLOCK_SIZE_BYTES * 8)
#define DRBG_ALGORITHM          TPM_ALG_AES
#if ((DRBG_KEY_SIZE_BITS % 8) != 0) || ((DRBG_IV_SIZE_BITS % 8) != 0)
#error "Key size and IV for DRBG must be even multiples of 8"
#endif
#if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
#error "Key size for DRBG must be even multiple of the cypher block size"
#endif
typedef UINT32    DRBG_SEED[(DRBG_KEY_SIZE_BITS + DRBG_IV_SIZE_BITS) / 32];
typedef struct {
    UINT64      reseedCounter;
    UINT32      magic;
    DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
    UINT32      lastValue[4];   // used when the TPM does continuous self-test
                                // for FIPS compliance of DRBG
} DRBG_STATE, *pDRBG_STATE;
```

B.3.5. Asymmetric Structures and Values

```
#ifdef TPM_ALG_ECC
```

B.3.6. ECC-related Structures

This structure replicates the structure definition in TPM_Types.h. It is duplicated to avoid inclusion of all of TPM_Types.h This structure is similar to the RSA_KEY structure below. The purpose of these structures is to reduce the overhead of a function call and to make the code less dependent on key types as much as possible.

```
typedef struct {
    UINT32                  curveID;       // The curve identifier
    TPMS_ECC_POINT         *publicPoint;   // Pointer to the public point
    TPM2B_ECC_PARAMETER    *privateKey;    // Pointer to the private key
} ECC_KEY;
#endif // TPM_ALG_ECC
#ifdef TPM_ALG_RSA
```

B.3.7. RSA-related Structures

This structure is a succinct representation of the cryptographic components of an RSA key.

```
typedef struct {
    UINT32        exponent;      // The public exponent pointer
    TPM2B        *publicKey;     // Pointer to the public modulus
    TPM2B        *privateKey;    // The private exponent (not a prime)
} RSA_KEY;
#endif // TPM_ALG_RSA
#ifdef TPM_ALG_RSA
#   ifdef TPM_ALG_ECC
#       if   MAX_RSA_KEY_BYTES > MAX_ECC_KEY_BYTES
#           define  MAX_NUMBER_SIZE         MAX_RSA_KEY_BYTES
#       else
#           define  MAX_NUMBER_SIZE         MAX_ECC_KEY_BYTES
#       endif
#   else // RSA but no ECC
#       define MAX_NUMBER_SIZE              MAX_RSA_KEY_BYTES
#   endif
#elif defined TPM_ALG_ECC
#   define MAX_NUMBER_SIZE                 MAX_ECC_KEY_BYTES
#else
#   error No assymmetric algorithm implemented.
#endif
typedef INT16     CRYPT_RESULT;
#define CRYPT_RESULT_MIN    INT16_MIN
#define CRYPT_RESULT_MAX    INT16_MAX
```


Table B.1

< 0	**recoverable error**
0	success
> 0	command specific return value (generally a digest size)

```
#define CRYPT_FAIL          ((CRYPT_RESULT)  1)
#define CRYPT_SUCCESS       ((CRYPT_RESULT)  0)
#define CRYPT_NO_RESULT     ((CRYPT_RESULT) -1)
#define CRYPT_SCHEME        ((CRYPT_RESULT) -2)
#define CRYPT_PARAMETER     ((CRYPT_RESULT) -3)
```



```
#define CRYPT_UNDERFLOW     ((CRYPT_RESULT) -4)
#define CRYPT_POINT         ((CRYPT_RESULT) -5)
#define CRYPT_CANCEL        ((CRYPT_RESULT) -6)
typedef UINT64              HASH_CONTEXT[MAX_HASH_STATE_SIZE/sizeof(UINT64)];
#include    "CpriCryptPri_fp.h"
#ifdef  TPM_ALG_ECC
#   include "CpriDataEcc.h"
#   include "CpriECC_fp.h"
#endif
#include    "MathFunctions_fp.h"
#include    "CpriRNG_fp.h"
#include    "CpriHash_fp.h"
#include    "CpriSym_fp.h"
#ifdef  TPM_ALG_RSA
#   include    "CpriRSA_fp.h"
#endif
#endif // !_CRYPT_PRI_H
```

B.4 OsslCryptoEngine.h

B.4.1. Introduction

This is the header file used by the components of the CryptoEngine(). This file should not be included in any file other than the files in the crypto engine.

Vendors may replace the implementation in this file by a local crypto engine. The implementation in this file is based on OpenSSL() library. Integer format: the big integers passed in/out the function interfaces in this library by a byte buffer (BYTE *) adopt the same format used in ISO/IEC 11889-1: An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer.

B.4.2. Defines

```
#ifndef _OSSL_CRYPTO_ENGINE_H
#define _OSSL_CRYPTO_ENGINE_H
#include <openssl/aes.h>
#include <openssl/camellia.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/ec.h>
#include <openssl/rand.h>
#include <openssl/bn.h>
#include <openSSL/ec_lcl.h>
#define     CRYPTO_ENGINE
#include "CryptoEngine.h"
#include "CpriMisc_fp.h"
#define MAX_ECC_PARAMETER_BYTES 32
#define MAX_2B_BYTES MAX((MAX_RSA_KEY_BYTES * ALG_RSA),            \
                         MAX((MAX_ECC_PARAMETER_BYTES * ALG_ECC),  \
                             MAX_DIGEST_SIZE))
#define assert2Bsize(a) pAssert((a).size <= <K>sizeof((a).buffer))
#ifdef TPM_ALG_RSA
#   ifdef    RSA_KEY_SIEVE
#       include     "RsaKeySieve.h"
#       include     "RsaKeySieve_fp.h"
#   endif
#   include     "CpriRSA_fp.h"
#endif
```

This is a structure to hold the parameters for the version of KDFa() used by the CryptoEngine(). This structure allows the state to be passed between multiple functions that use the same pseudo-random sequence.

```
typedef struct {
    CPRI_HASH_STATE          iPadCtx;
    CPRI_HASH_STATE          oPadCtx;
    TPM2B                   *extra;
    UINT32                  *outer;
    TPM_ALG_ID               hashAlg;
    UINT16                   keySizeInBits;
} KDFa_CONTEXT;
#endif // _OSSL_CRYPTO_ENGINE_H
```

B.5 MathFunctions.c

B.5.1. Introduction

This file contains implementation of some of the big number primitives. This is used in order to reduce the overhead in dealing with data conversions to standard big number format.

The simulator code uses the canonical form whenever possible in order to make the code in ISO/IEC 11889-3 more accessible. The canonical data formats are simple and not well suited for complex big number computations. This library provides functions that are found in typical big number libraries but they are written to handle the canonical data format of the reference TPM.

In some cases, data is converted to a big number format used by a standard library, such as OpenSSL(). This is done when the computations are complex enough warrant conversion. Vendors may replace the implementation in this file with a library that provides equivalent functions. A vendor may also rewrite the TPM code so that it uses a standard big number format instead of the canonical form and use the standard libraries instead of the code in this file.

The implementation in this file makes use of the OpenSSL() library.

Integer format: integers passed through the function interfaces in this library adopt the same format used in ISO/IEC 11889-1 that states:

"An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

An additional value is needed to indicate the number of significant bytes.

```
#include "OsslCryptoEngine.h"
```

B.5.2. Externally Accessible Functions

B.5.2.1. _math__Normalize2B()

This function will normalize the value in a TPM2B. If there are **leading** bytes of zero, the first non-zero byte is shifted up.

Table B.2

Return Value	Meaning
0	no significant bytes, value is zero
>0	number of significant bytes

```
LIB_EXPORT UINT16
_math__Normalize2B(
    TPM2B           *b              // IN/OUT: number to normalize
    )
{
    UINT16      from;
    UINT16      to;
    UINT16      size = b->size;

    for(from = 0; b->buffer[from] == 0 && from < size; from++);
    b->size -= from;
    for(to = 0; from < size; to++, from++ )
        b->buffer[to] = b->buffer[from];
    return b->size;
}
```

B.5.2.2. _math__Denormalize2B()

This function is used to adjust a TPM2B so that the number has the desired number of bytes. This is accomplished by adding bytes of zero at the start of the number.

Table B.3

Return Value	Meaning
TRUE	number de-normalized
FALSE	number already larger than the desired size

```
LIB_EXPORT BOOL
_math__Denormalize2B(
    TPM2B           *in,            // IN:OUT TPM2B number to de-normalize
    UINT32           size           // IN: the desired size
    )
{
    UINT32       to;
    UINT32       from;
    // If the current size is greater than the requested size, see if this can be
    // normalized to a value smaller than the requested size and then de-normalize
    if(in->size > size)
    {
        _math__Normalize2B(in);
        if(in->size > size)
            return FALSE;
    }
    // If the size is already what is requested, leave
    if(in->size == size)
        return TRUE;

    // move the bytes to the 'right'
    for(from = in->size, to = size; from > 0;)
        in->buffer[--to] = in->buffer[--from];

    // 'to' will always be greater than 0 because we checked for equal above.
    for(; to > 0;)
        in->buffer[--to] = 0;

    in->size = (UINT16)size;
    return TRUE;
}
```


B.5.2.3. _math__sub()

This function to subtract one unsigned value from another *c* = *a* - *b*. *c* may be the same as *a* or *b*.

Table B.4

Return Value	Meaning
1	if (a > b) so no borrow
0	if (a = b) so no borrow and b == a
-1	if (a < b) so there was a borrow

```
LIB_EXPORT int
_math__sub(
    const UINT32     aSize,         // IN: size of a
    const BYTE      *a,             // IN: a
    const UINT32     bSize,         // IN: size of b
    const BYTE      *b,             // IN: b
```

```
    UINT16          *cSize,         // OUT: set to MAX(aSize, bSize)
    BYTE            *c              // OUT: the difference
    )
{
    int             borrow = 0;
    int             notZero = 0;
    int             i;
    int             i2;

    // set c to the longer of a or b
    *cSize = (UINT16)((aSize > bSize) ? aSize : bSize);
    // pick the shorter of a and b
    i = (aSize > bSize) ? bSize : aSize;
    i2 = *cSize - i;
    a = &a[aSize - 1];
    b = &b[bSize - 1];
    c = &c[*cSize - 1];
    for(; i > 0; i--)
    {
        borrow = *a-- - *b-- + borrow;
        *c-- = (BYTE)borrow;
        notZero = notZero || borrow;
        borrow >>= 8;
    }
    if(aSize > bSize)
    {
        for(;i2 > 0; i2--)
        {
            borrow = *a-- + borrow;
            *c-- = (BYTE)borrow;
            notZero = notZero || borrow;
            borrow >>= 8;
        }
    }
    else if(aSize < bSize)
    {
        for(;i2 > 0; i2--)
        {
            borrow = 0 - *b-- + borrow;
            *c-- = (BYTE)borrow;
            notZero = notZero || borrow;
            borrow >>= 8;
        }
    }
    // if there is a borrow, then b > a
    if(borrow)
        return -1;
    // either a > b or they are the same
    return notZero;
}
```


B.5.2.4. _math__Inc()

This function increments a large, big-endian number value by one.

Table B.5

Return Value	Meaning
0	result is zero
!0	result is not zero

```
LIB_EXPORT int
_math__Inc(
```

```
    UINT32           aSize,         // IN: size of a
    BYTE            *a              // IN: a
    )
{

    for(a = &a[aSize-1];aSize > 0; aSize--)
    {
        if((*a-- += 1) != 0)
            return 1;
    }
    return 0;
}
```

B.5.2.5. _math__Dec

This function decrements a large, ENDIAN value by one.

```
LIB_EXPORT void
_math__Dec(
    UINT32           aSize,         // IN: size of a
    BYTE            *a              // IN: a
    )
{
    for(a = &a[aSize-1]; aSize > 0; aSize--)
    {
        if((*a-- -= 1) != 0xff)
            return;
    }
    return;
}
```

B.5.2.6. _math__Mul()

This function is used to multiply two large integers: *p* = *a** *b*. If the size of *p* is not specified (*pSize* == NULL), the size of the results *p* is assumed to be *aSize* + *bSize* and the results are de-normalized so that the resulting size is exactly *aSize* + *bSize*. If *pSize* is provided, then the actual size of the result is returned. The initial value for *pSize* must be at least *aSize* + *pSize*.

Table B.6

Return Value	Meaning
< 0	indicates an error
>= 0	the size of the product

```
LIB_EXPORT int
_math__Mul(
    const UINT32     aSize,         // IN: size of a
    const BYTE      *a,             // IN: a
    const UINT32     bSize,         // IN: size of b
    const BYTE      *b,             // IN: b
    UINT32          *pSize,         // IN/OUT: size of the product
    BYTE            *p              // OUT: product. length of product = aSize +
                                    //     bSize
    )
{
    BIGNUM          *bnA;
    BIGNUM          *bnB;
    BIGNUM          *bnP;
    BN_CTX          *context;
    int              retVal = 0;

```



```

    // First check that pSize is large enough if present
    if((pSize != NULL) && (*pSize < (aSize + bSize)))
        return CRYPT_PARAMETER;
    pAssert(pSize == NULL || *pSize <= MAX_2B_BYTES);
    //
    // Allocate space for BIGNUM context
    //
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    bnA = BN_CTX_get(context);
    bnB = BN_CTX_get(context);
    bnP = BN_CTX_get(context);
    if (bnP == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    // Convert the inputs to BIGNUMs
    //
    if (BN_bin2bn(a, aSize, bnA) == NULL || BN_bin2bn(b, bSize, bnB) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);

    // Perform the multiplication
    //
    if (BN_mul(bnP, bnA, bnB, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);


    // If the size of the results is allowed to float, then set the return
    // size. Otherwise, it might be necessary to de-normalize the results
    retVal = BN_num_bytes(bnP);
    if(pSize == NULL)
    {
        BN_bn2bin(bnP, &p[aSize + bSize - retVal]);
        memset(p, 0, aSize + bSize - retVal);
        retVal = aSize + bSize;
    }
    else
    {
        BN_bn2bin(bnP, p);
        *pSize = retVal;
    }

    BN_CTX_end(context);
    BN_CTX_free(context);
    return retVal;
}
```


B.5.2.7. _math__Div()

Divide an integer (*n*) by an integer (*d*) producing a quotient (*q*) and a remainder (*r*). If *q* or *r* is not needed, then the pointer to them may be set to NULL.

Table B.7

Return Value	Meaning
CRYPT_SUCCESS	operation complete
CRYPT_UNDERFLOW	*q* or *r* is too small to receive the result

```
LIB_EXPORT CRYPT_RESULT
_math__Div(
    const TPM2B     *n,             // IN: numerator
    const TPM2B     *d,             // IN: denominator
```

```
    TPM2B           *q,             // OUT: quotient
    TPM2B           *r              // OUT: remainder
    )
{
    BIGNUM          *bnN;
    BIGNUM          *bnD;
    BIGNUM          *bnQ;
    BIGNUM          *bnR;
    BN_CTX          *context;
    CRYPT_RESULT     retVal = CRYPT_SUCCESS;

    // Get structures for the big number representations
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnN = BN_CTX_get(context);
    bnD = BN_CTX_get(context);
    bnQ = BN_CTX_get(context);
    bnR = BN_CTX_get(context);

    // Errors in BN_CTX_get() are sticky so only need to check the last allocation
    if (   bnR == NULL
        || BN_bin2bn(n->buffer, n->size, bnN) == NULL
        || BN_bin2bn(d->buffer, d->size, bnD) == NULL)
            FAIL(FATAL_ERROR_INTERNAL);

    // Check for divide by zero.
    if(BN_num_bits(bnD) == 0)
        FAIL(FATAL_ERROR_DIVIDE_ZERO);

    // Perform the division
    if (BN_div(bnQ, bnR, bnN, bnD, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);


    // Convert the BIGNUM result back to our format
    if(q != NULL)   // If the quotient is being returned
    {
        if(!BnTo2B(q, bnQ, q->size))
        {
            retVal = CRYPT_UNDERFLOW;
            goto Done;
        }
     }
    if(r != NULL)   // If the remainder is being returned
    {
        if(!BnTo2B(r, bnR, r->size))
            retVal = CRYPT_UNDERFLOW;
    }

Done:
    BN_CTX_end(context);
    BN_CTX_free(context);

    return retVal;
}
```


B.5.2.8. _math__uComp()

This function compare two unsigned values.

Table B.8

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
LIB_EXPORT int
_math__uComp(
    const UINT32     aSize,          // IN: size of a
    const BYTE      *a,              // IN: a
    const UINT32     bSize,          // IN: size of b
    const BYTE      *b               // IN: b
    )
{
    int              borrow = 0;
    int              notZero = 0;
    int              i;
    // If a has more digits than b, then a is greater than b if
    // any of the more significant bytes is non zero
    if((i = (int)aSize - (int)bSize) > 0)
        for(; i > 0; i--)
            if(*a++) // means a > b
                return 1;
    // If b has more digits than a, then b is greater if any of the
    // more significant bytes is non zero
    if(i < 0)  <Q>// Means that b is longer than a
        for(; i < 0; i++)
            if(*b++) // means that b > a
                return -1;
    // Either the vales are the same size or the upper bytes of a or b are
    // all zero, so compare the rest
    i = (aSize > bSize) ? bSize : aSize;
    a = &a[i-1];
    b = &b[i-1];
    for(; i > 0; i--)
    {
        borrow = *a-- - *b-- + borrow;
        notZero = notZero || borrow;
        borrow >>= 8;
    }
    // if there is a borrow, then b > a
    if(borrow)
        return -1;
    // either a > b or they are the same
    return notZero;
}
```


B.5.2.9. _math__Comp()

Compare two signed integers:

Table B.9

Return Value	Meaning
1	if a > b
0	if a = b
-1	if a < b

```
LIB_EXPORT int
```

```
_math__Comp(
    const UINT32     aSize,          // IN: size of a
    const BYTE      *a,              // IN: a buffer
    const UINT32     bSize,          // IN: size of b
    const BYTE      *b               // IN: b buffer
    )
{
    int        signA, signB;       // sign of a and b

    // For positive or 0, sign_a is 1
    // for negative, sign_a is 0
    signA = ((a[0] & 0x80) == 0) ? 1 : 0;

    // For positive or 0, sign_b is 1
    // for negative, sign_b is 0
    signB = ((b[0] & 0x80) == 0) ? 1 : 0;

    if(signA != signB)
    {
        return signA - signB;
    }

    if(signA == 1)
        // do unsigned compare function
        return _math__uComp(aSize, a, bSize, b);
    else
        // do unsigned compare the other way
        return 0 - _math__uComp(aSize, a, bSize, b);
}
```

B.5.2.10. _math__ModExp

This function is used to do modular exponentiation in support of RSA. The most typical uses are: *c* = *m*^*e* mod *n* (RSA encrypt) and *m* = *c*^*d* mod *n* (RSA decrypt). When doing decryption, the *e* parameter of the function will contain the private exponent *d* instead of the public exponent *e*.

If the results will not fit in the provided buffer, an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that *m* be less than *n*.

Table B.10

Return Value	Meaning
CRYPT_SUCCESS	exponentiation succeeded
CRYPT_PARAMETER	number to exponentiate is larger than the modulus
CRYPT_UNDERFLOW	result will not fit into the provided buffer

```
LIB_EXPORT CRYPT_RESULT
_math__ModExp(
    UINT32           cSize,          // IN: size of the results
    BYTE            *c,              // OUT: results buffer
    const UINT32     mSize,          // IN: size of number to be exponentiated
    const BYTE      *m,              // IN: number to be exponentiated
    const UINT32     eSize,          // IN: size of power
    const BYTE      *e,              // IN: power
    const UINT32     nSize,          // IN: modulus size
    const BYTE      *n               // IN: modulus
    )
{
    CRYPT_RESULT     retVal = CRYPT_SUCCESS;
    BN_CTX          *context;
```



```
    BIGNUM          *bnC;
    BIGNUM          *bnM;
    BIGNUM          *bnE;
    BIGNUM          *bnN;
    INT32            i;

    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnC = BN_CTX_get(context);
    bnM = BN_CTX_get(context);
    bnE = BN_CTX_get(context);
    bnN = BN_CTX_get(context);

    // Errors for BN_CTX_get are sticky so only need to check last allocation
    if(bnN == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    //convert arguments
    if (   BN_bin2bn(m, mSize, bnM) == NULL
        || BN_bin2bn(e, eSize, bnE) == NULL
        || BN_bin2bn(n, nSize, bnN) == NULL)
            FAIL(FATAL_ERROR_INTERNAL);

    // Don't do exponentiation if the number being exponentiated is
    // larger than the modulus.
    if(BN_ucmp(bnM, bnN) >= 0)
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
    // Perform the exponentiation
    if(!(BN_mod_exp(bnC, bnM, bnE, bnN, context)))
        FAIL(FATAL_ERROR_INTERNAL);

    // Convert the results
    // Make sure that the results will fit in the provided buffer.
    if((unsigned)BN_num_bytes(bnC) > cSize)
    {
        retVal = CRYPT_UNDERFLOW;
        goto Cleanup;
    }
    i = cSize - BN_num_bytes(bnC);
    BN_bn2bin(bnC, &c[i]);
    memset(c, 0, i);

Cleanup:
    // Free up allocated BN values
    BN_CTX_end(context);
    BN_CTX_free(context);
    return retVal;
}
```

B.5.2.11. _math__IsPrime()

Check if an 32-bit integer is a prime.

Table B.11

Return Value	Meaning
TRUE	if the integer is probably a prime
FALSE	if the integer is definitely not a prime

```
LIB_EXPORT BOOL
_math__IsPrime(
    const UINT32     prime
    )
{
    int     isPrime;
    BIGNUM  *p;

    // Assume the size variables are not overflow, which should not happen in
    // the contexts that this function will be called.
    if((p = BN_new()) == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    if(!BN_set_word(p, prime))
        FAIL(FATAL_ERROR_INTERNAL);

    //
    // BN_is_prime returning -1 means that it ran into an error.
    // It should only return 0 or 1
    //
    if((isPrime = BN_is_prime_ex(p, BN_prime_checks, NULL, NULL)) < 0)
        FAIL(FATAL_ERROR_INTERNAL);

    if(p != NULL)
        BN_clear_free(p);
    return (isPrime == 1);
}
```

B.6 CpriCryptPri.c

B.6.1. Introduction

This file contains the interface to the initialization, startup and shutdown functions of the crypto library.

B.6.2. Includes and Locals

```
 #include "OsslCryptoEngine.h"
static void Trap(const char *function, int line, int code);
FAIL_FUNCTION       TpmFailFunction = (FAIL_FUNCTION)&Trap;
```

B.6.3. Functions

B.6.3.1. TpmFail()

This is a shim function that is called when a failure occurs. It simply relays the call to the callback pointed to by TpmFailFunction(). It is only defined for the sake of NO_RETURN specifier that cannot be added to a function pointer with some compilers.

```
void
TpmFail(
    const char          *function,
    int                  line,
    int                  code)
{
    TpmFailFunction(function, line, code);
}
```

B.6.3.2. FAILURE_TRAP()

This function is called if the caller to _cpri__InitCryptoUnits() doesn't provide a call back address.

```
static void
Trap(
    const char       *function,
    int               line,
    int               code
    )
{
    UNREFERENCED(function);
    UNREFERENCED(line);
    UNREFERENCED(code);
    abort();
}
```

B.6.3.3. _cpri__InitCryptoUnits()

This function calls the initialization functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called as a result of _TPM_Init(). The parameter to this function is a call back function it TMP. lib that is called when the crypto engine has a failure.

```
LIB_EXPORT CRYPT_RESULT
_cpri__InitCryptoUnits(
    FAIL_FUNCTION    failFunction
    )
{
```

```
    TpmFailFunction = failFunction;

    _cpri__RngStartup();
    _cpri__HashStartup();
    _cpri__SymStartup();

#ifdef TPM_ALG_RSA
    _cpri__RsaStartup();
#endif

#ifdef TPM_ALG_ECC
    _cpri__EccStartup();
#endif

    return CRYPT_SUCCESS;
}
```

B.6.3.4. _cpri__StopCryptoUnits()

This function calls the shutdown functions of the other crypto modules that are part of the crypto engine for this implementation.

```
LIB_EXPORT void
_cpri__StopCryptoUnits(
    void
    )
{
    return;
}
```

B.6.3.5. _cpri__Startup()

This function calls the startup functions of the other crypto modules that are part of the crypto engine for this implementation. This function should be called during processing of TPM2_Startup().

```
LIB_EXPORT BOOL
_cpri__Startup(
    void
    )
{

    return(    _cpri__HashStartup()
            && _cpri__RngStartup()
#ifdef TPM_ALG_RSA
            && _cpri__RsaStartup()
#endif // TPM_ALG_RSA
#ifdef TPM_ALG_ECC
            && _cpri__EccStartup()
#endif // TPM_ALG_ECC
            && _cpri__SymStartup());
}
```


B.7 CpriRNG.c

B.7.1. Introduction

This file contains the interface to the OpenSSL() random number functions.

B.7.2. Defines

```
//#define __TPM_RNG_FOR_DEBUG__
```

B.7.3. Includes and Values

```
#include "OsslCryptoEngine.h"
int          s_entropyFailure;
```

B.7.4. Functions

B.7.4.1. _cpri__RngStartup()

This function is called to initialize the random number generator. It collects entropy from the platform to seed the OpenSSL() random number generator.

```
LIB_EXPORT BOOL
_cpri__RngStartup(void)
{
    UINT32        entropySize;
    BYTE          entropy[MAX_RNG_ENTROPY_SIZE];
    INT32         returnedSize = 0;

    // Initialize the entropy source
    s_entropyFailure = FALSE;
    _plat__GetEntropy(NULL, 0);

    // Collect entropy until we have enough
    for(entropySize  = 0;
        entropySize < MAX_RNG_ENTROPY_SIZE && returnedSize >= 0;
        entropySize += returnedSize)
    {
        returnedSize = _plat__GetEntropy(&entropy[entropySize],
                                    MAX_RNG_ENTROPY_SIZE - entropySize);
    }
    // Got some entropy on the last call and did not get an error
    if(returnedSize > 0)
    {
        // Seed OpenSSL with entropy
        RAND_seed(entropy, entropySize);
    }
    else
    {
        s_entropyFailure = TRUE;
    }
    return s_entropyFailure == FALSE;
}
```

B.7.4.2. _cpri__DrbgGetPutState()

This function is used to set the state of the RNG *(direction* == PUT_STATE) or to recover the state of the RNG *(direction* == GET_STATE).

NOTE This not currently supported on OpenSSL() version.

```
LIB_EXPORT CRYPT_RESULT
_cpri__DrbgGetPutState(
    GET_PUT          direction,
    int              bufferSize,
    BYTE            *buffer
    )
{
    UNREFERENCED_PARAMETER(direction);
    UNREFERENCED_PARAMETER(bufferSize);
    UNREFERENCED_PARAMETER(buffer);

    return CRYPT_SUCCESS;       // Function is not implemented
}
```

B.7.4.3. _cpri__StirRandom()

This function is called to add external entropy to the OpenSSL() random number generator.

```
LIB_EXPORT CRYPT_RESULT
_cpri__StirRandom(
    INT32            entropySize,
    BYTE            *entropy
    )
{
    if (entropySize >= 0)
    {
        RAND_add((const void *)entropy, (int) entropySize, 0.0);

    }
    return CRYPT_SUCCESS;
}
```

B.7.4.4. _cpri__GenerateRandom()

This function is called to get a string of random bytes from the OpenSSL() random number generator. The return value is the number of bytes placed in the buffer. If the number of bytes returned is not equal to the number of bytes requested (*randomSize*) it is indicative of a failure of the OpenSSL() random number generator and is probably fatal.

```
LIB_EXPORT UINT16
_cpri__GenerateRandom(
    INT32            randomSize,
    BYTE            *buffer
    )
{
    //
    // We don't do negative sizes or ones that are too large
    if (randomSize < 0 || randomSize > UINT16_MAX)
        return 0;
    // RAND_bytes uses 1 for success and we use 0
    if(RAND_bytes(buffer, randomSize) == 1)
        return (UINT16)randomSize;
    else
        return 0;
}
```

B.7.4.5. _cpri__GenerateSeededRandom()

This function is used to generate a pseudo-random number from some seed values. This function returns the same result each time it is called with the same parameters

```
LIB_EXPORT UINT16
_cpri__GenerateSeededRandom(
    INT32             randomSize,    // IN: the size of the request
    BYTE             *random,        // OUT: receives the data
    TPM_ALG_ID        hashAlg,       // IN: used by KDF version but not here
    TPM2B            *seed,          // IN: the seed value
    const char       *label,         // IN: a label string (optional)
    TPM2B            *partyU,        // IN: other data (oprtional)
    TPM2B            *partyV         // IN: still more (optional)
    )
{

    return (_cpri__KDFa(hashAlg, seed, label, partyU, partyV,
                        randomSize * 8, random, NULL, FALSE));
}
#endif  //%
```

B.8 CpriHash.c

B.8.1. Description

This file contains implementation of cryptographic functions for hashing.

B.8.2. Includes, Defines, and Types

```
#include    "OsslCryptoEngine.h"
#include    "CpriHashData.c"
#define OSSL_HASH_STATE_DATA_SIZE    (MAX_HASH_STATE_SIZE - 8)
typedef struct {
    union   {
        EVP_MD_CTX  context;
        BYTE        data[OSSL_HASH_STATE_DATA_SIZE];
    } u;
    INT16           copySize;
} OSSL_HASH_STATE;
```

Temporary aliasing of SM3 to SHA256 until SM3 is available

```
#define EVP_sm3_256 EVP_sha256
```

B.8.3. Static Functions

B.8.3.1. GetHashServer()

This function returns the address of the hash server function

```
static EVP_MD *
GetHashServer(
    TPM_ALG_ID   hashAlg
)
{
    switch (hashAlg)
    {
#ifdef TPM_ALG_SHA1
    case TPM_ALG_SHA1:
        return (EVP_MD *)EVP_sha1();
        break;
#endif
#ifdef TPM_ALG_SHA256
    case TPM_ALG_SHA256:
        return (EVP_MD *)EVP_sha256();
        break;
#endif
#ifdef TPM_ALG_SHA384
    case TPM_ALG_SHA384:
        return (EVP_MD *)EVP_sha384();
        break;
#endif
#ifdef TPM_ALG_SHA512
    case TPM_ALG_SHA512:
        return (EVP_MD *)EVP_sha512();
        break;
#endif
#ifdef TPM_ALG_SM3_256
    case TPM_ALG_SM3_256:
        return (EVP_MD *)EVP_sm3_256();
        break;
```



```
#endif
    case TPM_ALG_NULL:
        return NULL;
    default:
        FAIL(FATAL_ERROR_INTERNAL);
    }
}
```

B.8.3.2. MarshalHashState()

This function copies an OpenSSL() hash context into a caller provided buffer.

Table B.12

Return Value	Meaning
> 0	the number of bytes of buf used.

```
static UINT16
MarshalHashState(
    EVP_MD_CTX      *ctxt,          // IN: Context to marshal
    BYTE            *buf            // OUT: The buffer that will receive the
                                    //     context. This buffer is at least
                                    //     MAX_HASH_STATE_SIZE bytes
    )
{
    // make sure everything will fit
    pAssert(ctxt->digest->ctx_size <= OSSL_HASH_STATE_DATA_SIZE);

    // Copy the context data
    memcpy(buf, (void*) ctxt->md_data, ctxt->digest->ctx_size);

    return (UINT16)ctxt->digest->ctx_size;
}
```

B.8.3.3. GetHashState()

This function will unmarshal a caller provided buffer into an OpenSSL() hash context. The function returns the number of bytes copied (which may be zero).

```
static UINT16
GetHashState(
    EVP_MD_CTX      *ctxt,          // OUT: The context structure to receive the
                                    //     result of unmarshaling.
    TPM_ALG_ID       algType,       // IN: The hash algorithm selector
    BYTE            *buf            // IN: Buffer containing marshaled hash data
    )
{
    EVP_MD          *evpmdAlgorithm = NULL;

    pAssert(ctxt != NULL);

    EVP_MD_CTX_init(ctxt);

    evpmdAlgorithm = GetHashServer(algType);
    if(evpmdAlgorithm == NULL)
        return 0;

    // This also allocates the ctxt->md_data
    if((EVP_DigestInit_ex(ctxt, evpmdAlgorithm, NULL)) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    pAssert(ctxt->digest->ctx_size < <K>sizeof(ALIGNED_HASH_STATE));
```

```
    memcpy(ctxt->md_data, buf, ctxt->digest->ctx_size);
    return (UINT16)ctxt->digest->ctx_size;
}
```

B.8.3.4. GetHashInfoPointer()

This function returns a pointer to the hash info for the algorithm. If the algorithm is not supported, function returns a pointer to the data block associated with TPM_ALG_NULL.

```
static const HASH_INFO *
GetHashInfoPointer(
    TPM_ALG_ID      hashAlg
    )
{
    UINT32 i, tableSize;

    // Get the table size of g_hashData
    tableSize = sizeof(g_hashData) / sizeof(g_hashData[0]);

    for(i = 0; i < tableSize - 1; i++)
    {
        if(g_hashData[i].alg == hashAlg)
            return &g_hashData[i];
    }
    return &g_hashData[tableSize-1];
}
```

B.8.4. Hash Functions

B.8.4.1. _cpri__HashStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
LIB_EXPORT BOOL
_cpri__HashStartup(
    void
    )
{
    // On startup, make sure that the structure sizes are compatible. It would
    // be nice if this could be done at compile time but I couldn't figure it out.
    CPRI_HASH_STATE *cpriState = NULL;
//    NUMBYTES        evpCtxSize = sizeof(EVP_MD_CTX);
    NUMBYTES        cpriStateSize = sizeof(cpriState->state);
//    OSSL_HASH_STATE *osslState;
    NUMBYTES        osslStateSize = sizeof(OSSL_HASH_STATE);
//    int             dataSize = sizeof(osslState->u.data);
    pAssert(cpriStateSize >= osslStateSize);

    return TRUE;
}
```

B.8.4.2. _cpri__GetHashAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and and *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Table B.13

Return Value	Meaning
TPM_ALG_xxx()	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```
LIB_EXPORT TPM_ALG_ID
_cpri__GetHashAlgByIndex(
    UINT32          index           // IN: the index
    )
{
    if(index >= HASH_COUNT)
        return TPM_ALG_NULL;
    return g_hashData[index].alg;
}
```

B.8.4.3. _cpri__GetHashBlockSize()

Returns the size of the block used for the hash.

Table B.14

Return Value	Meaning
< 0	the algorithm is not a supported hash
>=	the digest size (0 for TPM_ALG_NULL)

```
LIB_EXPORT UINT16
_cpri__GetHashBlockSize(
    TPM_ALG_ID      hashAlg         // IN: hash algorithm to look up
    )
{
    return GetHashInfoPointer(hashAlg)->blockSize;
}
```

B.8.4.4. _cpri__GetHashDER

This function returns a pointer to the DER string for the algorithm and indicates its size.

```
LIB_EXPORT UINT16
_cpri__GetHashDER(
    TPM_ALG_ID      hashAlg,        // IN: the algorithm to look up
    const BYTE     **p
    )
{
    const HASH_INFO      *q;
    q = GetHashInfoPointer(hashAlg);
    *p = &q->der[0];
    return q->derSize;
}
```

B.8.4.5. _cpri__GetDigestSize()

Gets the digest size of the algorithm. The algorithm is required to be supported.

Table B.15

Return Value	Meaning
=0	the digest size for TPM_ALG_NULL
>0	the digest size of a hash algorithm

```
LIB_EXPORT UINT16
_cpri__GetDigestSize(
    TPM_ALG_ID       hashAlg        // IN: hash algorithm to look up
    )
{
    return GetHashInfoPointer(hashAlg)->digestSize;
}
```

B.8.4.6. _cpri__GetContextAlg()

This function returns the algorithm associated with a hash context

```
LIB_EXPORT TPM_ALG_ID
_cpri__GetContextAlg(
    CPRI_HASH_STATE     *hashState      // IN: the hash context
    )
{
    return hashState->hashAlg;
}
```

B.8.4.7. _cpri__CopyHashState

This function is used to **clone** a CPRI_HASH_STATE. The return value is the size of the state.

```
LIB_EXPORT UINT16
_cpri__CopyHashState (
    CPRI_HASH_STATE     *out,           // OUT: destination of the state
    CPRI_HASH_STATE     *in             // IN: source of the state
    )
{
    OSSL_HASH_STATE    *i = (OSSL_HASH_STATE *)&in->state;
    OSSL_HASH_STATE    *o = (OSSL_HASH_STATE *)&out->state;
    pAssert(sizeof(i) <= <K>sizeof(in->state));

    EVP_MD_CTX_init(&o->u.context);
    EVP_MD_CTX_copy_ex(&o->u.context, &i->u.context);
    o->copySize = i->copySize;
    out->hashAlg = in->hashAlg;
    return sizeof(CPRI_HASH_STATE);
}
```


B.8.4.8. _cpri__StartHash()

Functions starts a hash stack Start a hash stack and returns the digest size. As a side effect, the value of *stateSize* in *hashState* is updated to indicate the number of bytes of state that were saved. This function calls GetHashServer() and that function will put the TPM into failure mode if the hash algorithm is not supported.

Table B.16

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```
LIB_EXPORT UINT16
_cpri__StartHash(
    TPM_ALG_ID          hashAlg,        // IN: hash algorithm
    BOOL                sequence,       // IN: TRUE if the state should be saved
    CPRI_HASH_STATE    *hashState       // OUT: the state of hash stack.
    )
{
    EVP_MD_CTX        localState;
    OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
    BYTE             *stateData = state->u.data;
    EVP_MD_CTX       *context;
    EVP_MD           *evpmdAlgorithm = NULL;
    UINT16            retVal = 0;

    if(sequence)
        context = &localState;
    else
        context = &state->u.context;

    hashState->hashAlg = hashAlg;

    EVP_MD_CTX_init(context);
    evpmdAlgorithm = GetHashServer(hashAlg);
    if(evpmdAlgorithm == NULL)
        goto Cleanup;

    if(EVP_DigestInit_ex(context, evpmdAlgorithm, NULL) != 1)
        FAIL(FATAL_ERROR_INTERNAL);
    retVal = (CRYPT_RESULT)EVP_MD_CTX_size(context);

Cleanup:
    if(retVal > 0)
    {
        if (sequence)
        {
            if((state->copySize = MarshalHashState(context, stateData)) == 0)
            {
                // If MarshalHashState returns a negative number, it is an error
                // code and not a hash size so copy the error code to be the return
                // from this function and set the actual stateSize to zero.
                retVal = state->copySize;
                state->copySize = 0;
            }
            // Do the cleanup
            EVP_MD_CTX_cleanup(context);
        }
        else
            state->copySize = -1;
    }
    else
        state->copySize = 0;
    return retVal;
}
```


B.8.4.9. _cpri__UpdateHash()

Add data to a hash or HMAC stack.

```
LIB_EXPORT void
_cpri__UpdateHash(
    CPRI_HASH_STATE     *hashState,      // IN: the hash context information
    UINT32               dataSize,       // IN: the size of data to be added to the
                                         //     digest
    BYTE                *data            // IN: data to be hashed
    )
{
    EVP_MD_CTX       localContext;
    OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
    BYTE            *stateData = state->u.data;
    EVP_MD_CTX      *context;
    CRYPT_RESULT     retVal = CRYPT_SUCCESS;

    // If there is no context, return
    if(state->copySize == 0)
        return;
    if(state->copySize > 0)
    {
        context = &localContext;
        if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
            return;
    }
    else
        context = &state->u.context;

    if(EVP_DigestUpdate(context, data, dataSize) != 1)
        FAIL(FATAL_ERROR_INTERNAL);
    else if(   state->copySize > 0
            && (retVal= MarshalHashState(context, stateData)) >= 0)
    {
        // retVal is the size of the marshaled data. Make sure that it is consistent
        // by ensuring that we didn't get more than allowed
        if(retVal < state->copySize)
            FAIL(FATAL_ERROR_INTERNAL);
        else
            EVP_MD_CTX_cleanup(context);
    }
    return;
}
```

B.8.4.10. _cpri__CompleteHash()

Complete a hash or HMAC computation. This function will place the smaller of *digestSize* or the size of the digest in *dOut*. The number of bytes in the placed in the buffer is returned. If there is a failure, the returned value is <= 0.

Table B.17

Return Value	Meaning
0	no data returned
> 0	the number of bytes in the digest

```
LIB_EXPORT UINT16
_cpri__CompleteHash(
    CPRI_HASH_STATE     *hashState,      // IN: the state of hash stack
    UINT32               dOutSize,       // IN: size of digest buffer
    BYTE                *dOut            // OUT: hash digest
    )
{
    EVP_MD_CTX       localState;
    OSSL_HASH_STATE *state = (OSSL_HASH_STATE *)&hashState->state;
    BYTE            *stateData = state->u.data;
```



```
    EVP_MD_CTX      *context;
    UINT16           retVal;
    int              hLen;
    BYTE             temp[MAX_DIGEST_SIZE];
    BYTE            *rBuffer = dOut;

    if(state->copySize == 0)
        return 0;
    if(state->copySize > 0)
    {
        context = &localState;
        if((retVal = GetHashState(context, hashState->hashAlg, stateData)) <= 0)
            goto Cleanup;
    }
    else
        context = &state->u.context;

    hLen = EVP_MD_CTX_size(context);
    if((unsigned)hLen > dOutSize)
        rBuffer = temp;
    if(EVP_DigestFinal_ex(context, rBuffer, NULL) == 1)
    {
        if(rBuffer != dOut)
        {
            if(dOut != NULL)
            {
                memcpy(dOut, temp, dOutSize);
            }
            retVal = (UINT16)dOutSize;
        }
        else
        {
            retVal = (UINT16)hLen;
        }
        state->copySize = 0;
    }
    else
    {
        retVal = 0; // Indicate that no data is returned
    }
Cleanup:
    EVP_MD_CTX_cleanup(context);
    return retVal;
}
```

B.8.4.11. _cpri__ImportExportHashState()

This function is used to import or export the hash state. This function would be called to export state when a sequence object was being prepared for export

```
LIB_EXPORT void
_cpri__ImportExportHashState(
    CPRI_HASH_STATE     *osslFmt,       // IN/OUT: the hash state formated for use
                                        //      by openSSL
    EXPORT_HASH_STATE   *externalFmt,   // IN/OUT: the exported hash state
    IMPORT_EXPORT        direction      //
    )
{
    UNREFERENCED_PARAMETER(direction);
    UNREFERENCED_PARAMETER(externalFmt);
    UNREFERENCED_PARAMETER(osslFmt);
    return;

#if 0
```

```
    if(direction == IMPORT_STATE)
    {
        // don't have the import export functions yet so just copy
        _cpri__CopyHashState(osslFmt, (CPRI_HASH_STATE *)externalFmt);
    }
    else
    {
        _cpri__CopyHashState((CPRI_HASH_STATE *)externalFmt, osslFmt);
    }
#endif
}
```

B.8.4.12. _cpri__HashBlock()

Start a hash, hash a single block, update *digest* and return the size of the results.

The **digestSize** parameter can be smaller than the digest. If so, only the more significant bytes are returned.

Table B.18

Return Value	Meaning
>= 0	number of bytes in *digest* (may be zero)

```
LIB_EXPORT UINT16
_cpri__HashBlock(
    TPM_ALG_ID       hashAlg,       // IN: The hash algorithm
    UINT32           dataSize,      // IN: size of buffer to hash
    BYTE            *data,          // IN: the buffer to hash
    UINT32           digestSize,    // IN: size of the digest buffer
    BYTE            *digest         // OUT: hash digest
    )
{
    EVP_MD_CTX       hashContext;
    EVP_MD          *hashServer = NULL;
    UINT16           retVal = 0;
    BYTE             b[MAX_DIGEST_SIZE]; // temp buffer in case digestSize not
    // a full digest
    unsigned int     dSize = _cpri__GetDigestSize(hashAlg);


    // If there is no digest to compute return
    if(dSize == 0)
        return 0;

    // After the call to EVP_MD_CTX_init(), will need to call EVP_MD_CTX_cleanup()
    EVP_MD_CTX_init(&hashContext);      // Initialize the local hash context
    hashServer = GetHashServer(hashAlg); // Find the hash server

    // It is an error if the digest size is non-zero but there is no server
    if(    (hashServer == NULL)
       || (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
       || (EVP_DigestUpdate(&hashContext, data, dataSize) != 1))
        FAIL(FATAL_ERROR_INTERNAL);
    else
    {
        // If the size of the digest produced (dSize) is larger than the available
        // buffer (digestSize), then put the digest in a temp buffer and only copy
        // the most significant part into the available buffer.
        if(dSize > digestSize)
        {
            if(EVP_DigestFinal_ex(&hashContext, b, &dSize) != 1)
                FAIL(FATAL_ERROR_INTERNAL);
            memcpy(digest, b, digestSize);
```



```
            retVal = (UINT16)digestSize;
        }
        else
        {
            if((EVP_DigestFinal_ex(&hashContext, digest, &dSize)) != 1)
                FAIL(FATAL_ERROR_INTERNAL);
            retVal = (UINT16) dSize;
        }
    }
    EVP_MD_CTX_cleanup(&hashContext);
    return retVal;
}
```

B.8.5. HMAC Functions

B.8.5.1. _cpri__StartHMAC

This function is used to start an HMAC using a temp hash context. The function does the initialization of the hash with the HMAC key XOR *iPad* and updates the HMAC key XOR *oPad*.

The function returns the number of bytes in a digest produced by *hashAlg*.

Table B.19

Return Value	Meaning
>= 0	number of bytes in digest produced by *hashAlg* (may be zero)

```
LIB_EXPORT UINT16
_cpri__StartHMAC(
    TPM_ALG_ID          hashAlg,        // IN: the algorithm to use
    BOOL                sequence,       // IN: indicates if the state should be
                                        //     saved
    CPRI_HASH_STATE    *state,          // IN/OUT: the state buffer
    UINT16              keySize,        // IN: the size of the HMAC key
    BYTE               *key,            // IN: the HMAC key
    TPM2B              *oPadKey         // OUT: the key prepared for the oPad round
    )
{
    CPRI_HASH_STATE  localState;
    UINT16           blockSize = _cpri__GetHashBlockSize(hashAlg);
    UINT16           digestSize;
    BYTE            *pb;        // temp pointer
    UINT32           i;

    // If the key size is larger than the block size, then the hash of the key
    // is used as the key
    if(keySize > blockSize)
    {
        // large key so digest
        if((digestSize = _cpri__StartHash(hashAlg, FALSE, &localState)) == 0)
            return 0;
        _cpri__UpdateHash(&localState, keySize, key);
        _cpri__CompleteHash(&localState, digestSize, oPadKey->buffer);
        oPadKey->size = digestSize;
    }
    else
    {
        // key size is ok
        memcpy(oPadKey->buffer, key, keySize);
        oPadKey->size = keySize;
    }
    // XOR the key with iPad (0x36)
    pb = oPadKey->buffer;
```

```
    for(i = oPadKey->size; i > 0; i--)
        *pb++ ^= 0x36;

    // if the keySize is smaller than a block, fill the rest with 0x36
    for(i = blockSize - oPadKey->size; i >  0; i--)
        *pb++ = 0x36;

    // Increase the oPadSize to a full block
    oPadKey->size = blockSize;

    // Start a new hash with the HMAC key
    // This will go in the caller's state structure and may be a sequence or not

    if((digestSize = _cpri__StartHash(hashAlg, sequence, state)) > 0)
    {

        _cpri__UpdateHash(state, oPadKey->size, oPadKey->buffer);

        // XOR the key block with 0x5c ^ 0x36
        for(pb = oPadKey->buffer, i = blockSize; i > 0; i--)
            *pb++ ^= (0x5c ^ 0x36);
    }

    return digestSize;
}
```

B.8.5.2. _cpri_CompleteHMAC()

This function is called to complete an HMAC. It will finish the current digest, and start a new digest. It will then add the *oPadKey* and the completed digest and return the results in *dOut*. It will not return more than *dOutSize* bytes.

Table B.20

Return Value	Meaning
>= 0	number of bytes in *dOut* (may be zero)

```
LIB_EXPORT UINT16
_cpri__CompleteHMAC(
    CPRI_HASH_STATE     *hashState,     // IN: the state of hash stack
    TPM2B               *oPadKey,       // IN: the HMAC key in oPad format
    UINT32               dOutSize,      // IN: size of digest buffer
    BYTE                *dOut           // OUT: hash digest
    )
{
    BYTE               digest[MAX_DIGEST_SIZE];
    CPRI_HASH_STATE *state = (CPRI_HASH_STATE *)hashState;
    CPRI_HASH_STATE  localState;
    UINT16             digestSize = _cpri__GetDigestSize(state->hashAlg);


    _cpri__CompleteHash(hashState, digestSize, digest);

    // Using the local hash state, do a hash with the oPad
    if(_cpri__StartHash(state->hashAlg, FALSE, &localState) != digestSize)
        return 0;

    _cpri__UpdateHash(&localState, oPadKey->size, oPadKey->buffer);
    _cpri__UpdateHash(&localState, digestSize, digest);
    return _cpri__CompleteHash(&localState, dOutSize, dOut);
}
```


B.8.6. Mask and Key Generation Functions

B.8.6.1. _crypi_MGF1()

This function performs MGF1 using the selected hash. MGF1 is T(n) = T(n-1) || H(seed || counter). This function returns the length of the mask produced which could be zero if the digest algorithm is not supported.

Table B.21

Return Value	Meaning
0	hash algorithm not supported
> 0	should be the same as *mSize*

```
LIB_EXPORT CRYPT_RESULT
_cpri__MGF1(
    UINT32           mSize,         // IN: length of the mask to be produced
    BYTE            *mask,          // OUT: buffer to receive the mask
    TPM_ALG_ID       hashAlg,       // IN: hash to use
    UINT32           sSize,         // IN: size of the seed
    BYTE            *seed           // IN: seed size
    )
{
    EVP_MD_CTX           hashContext;
    EVP_MD              *hashServer = NULL;
    CRYPT_RESULT         retVal = 0;
    BYTE                 b[MAX_DIGEST_SIZE]; // temp buffer in case mask is not an
    // even multiple of a full digest
    CRYPT_RESULT         dSize = _cpri__GetDigestSize(hashAlg);
    unsigned int         digestSize = (UINT32)dSize;
    UINT32               remaining;
    UINT32               counter;
    BYTE                 swappedCounter[4];

    // Parameter check
    if(mSize > (1024*16)) // Semi-arbitrary maximum
        FAIL(FATAL_ERROR_INTERNAL);

    // If there is no digest to compute return
    if(dSize <= 0)
        return 0;

    EVP_MD_CTX_init(&hashContext);     // Initialize the local hash context
    hashServer = GetHashServer(hashAlg); // Find the hash server
    if(hashServer == NULL)
        // If there is no server, then there is no digest
        return 0;

    for(counter = 0, remaining = mSize; remaining > 0; counter++)
    {
        // Because the system may be either Endian...
        UINT32_TO_BYTE_ARRAY(counter, swappedCounter);

        // Start the hash and include the seed and counter
        if(    (EVP_DigestInit_ex(&hashContext, hashServer, NULL) != 1)
            || (EVP_DigestUpdate(&hashContext, seed, sSize) != 1)
            || (EVP_DigestUpdate(&hashContext, swappedCounter, 4) != 1)
          )
            FAIL(FATAL_ERROR_INTERNAL);

        // Handling the completion depends on how much space remains in the mask
        // buffer. If it can hold the entire digest, put it there. If not
        // put the digest in a temp buffer and only copy the amount that
```

```
            // will fit into the mask buffer.
            if(remaining < (<K>unsigned)dSize)
            {
                if(EVP_DigestFinal_ex(&hashContext, b, &digestSize) != 1)
                    FAIL(FATAL_ERROR_INTERNAL);
                memcpy(mask, b, remaining);
                break;
            }
            else
            {
                if(EVP_DigestFinal_ex(&hashContext, mask, &digestSize) != 1)
                    FAIL(FATAL_ERROR_INTERNAL);
                remaining -= dSize;
                mask = &mask[dSize];
            }
            retVal = (CRYPT_RESULT)mSize;
        }

        EVP_MD_CTX_cleanup(&hashContext);
        return retVal;
    }
```

B.8.6.2. _cpri_KDFa()

This function performs the key generation according to ISO/IEC 11889-1.

This function returns the number of bytes generated which may be zero.

The *key* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than (2^18)-1 = 256K bits (32385 bytes).

The **once** parameter is set to allow incremental generation of a large value. If this flag is TRUE, **sizeInBits** will be used in the HMAC computation but only one iteration of the KDF is performed. This would be used for XOR obfuscation so that the mask value can be generated in digest-sized chunks rather than having to be generated all at once in an arbitrarily large buffer and then XORed() into the result. If **once** is TRUE, then **sizeInBits** must be a multiple of 8.

Any error in the processing of this command is considered fatal.

Table B.22

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
LIB_EXPORT UINT16
_cpri__KDFa(
    TPM_ALG_ID        hashAlg,       // IN: hash algorithm used in HMAC
    TPM2B            *key,           // IN: HMAC key
    const char       *label,         // IN: a 0-byte terminated label used in KDF
    TPM2B            *contextU,      // IN: context U
    TPM2B            *contextV,      // IN: context V
    UINT32            sizeInBits,    // IN: size of generated key in bits
    BYTE             *keyStream,     // OUT: key buffer
    UINT32           *counterInOut,  // IN/OUT: caller may provide the iteration
                                     //        counter for incremental operations to
                                     //        avoid large intermediate buffers.
    BOOL              once           // IN: TRUE if only one iteration is performed
                                     //        FALSE if iteration count determined by
                                     //        "sizeInBits"
    )
{
    UINT32                    counter = 0;      // counter value
```

```
    INT32                    lLen = 0;       // length of the label
    INT16                    hLen;           // length of the hash
    INT16                    bytes;          // number of bytes to produce
    BYTE                    *stream = keyStream;
    BYTE                     marshaledUint32[4];
    CPRI_HASH_STATE          hashState;
    TPM2B_MAX_HASH_BLOCK     hmacKey;

    pAssert(key != NULL && keyStream != NULL);
    pAssert(once == FALSE || (sizeInBits & 7) == 0);

    if(counterInOut != NULL)
        counter = *counterInOut;

    // Prepare label buffer.  Calculate its size and keep the last 0 byte
    if(label != NULL)
        for(lLen = 0; label[lLen++] != 0; );

    // Get the hash size.  If it is less than or 0, either the
    // algorithm is not supported or the hash is TPM_ALG_NULL
    // In either case the digest size is zero.  This is the only return
    // other than the one at the end. All other exits from this function
    // are fatal errors. After we check that the algorithm is supported
    // anything else that goes wrong is an implementation flaw.
    if((hLen = (INT16) _cpri__GetDigestSize(hashAlg)) == 0)
        return 0;

    // If the size of the request is larger than the numbers will handle,
    // it is a fatal error.
    pAssert(((sizeInBits + 7)/ 8) <= INT16_MAX);

    bytes = once ? hLen : (INT16)((sizeInBits + 7) / 8);

    // Generate required bytes
    for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
    {
        if(bytes < hLen)
            hLen = bytes;

        counter++;
        // Start HMAC
        if(_cpri__StartHMAC(hashAlg,
                            FALSE,
                            &hashState,
                            key->size,
                            &key->buffer[0],
                            &hmacKey.b)          <= 0)
            FAIL(FATAL_ERROR_INTERNAL);

        // Adding counter
        UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
        _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

        // Adding label
        if(label != NULL)
            _cpri__UpdateHash(&hashState,  lLen, (BYTE *)label);

        // Adding contextU
        if(contextU != NULL)
            _cpri__UpdateHash(&hashState, contextU->size, contextU->buffer);

        // Adding contextV
        if(contextV != NULL)
            _cpri__UpdateHash(&hashState, contextV->size, contextV->buffer);

        // Adding size in bits
```



```
            UINT32_TO_BYTE_ARRAY(sizeInBits, marshaledUint32);
            _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

            // Compute HMAC. At the start of each iteration, hLen is set
            // to the smaller of hLen and bytes. This causes bytes to decrement
            // exactly to zero to complete the loop
            _cpri__CompleteHMAC(&hashState, &hmacKey.b, hLen, stream);
        }

        // Mask off bits if the required bits is not a multiple of byte size
        if((sizeInBits % 8) != 0)
            keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
        if(counterInOut != NULL)
            *counterInOut = counter;
        return (CRYPT_RESULT)((sizeInBits + 7)/8);
    }
```

B.8.6.3. _cpri__KDFe()

KDFe() as defined in ISO/IEC 11889-1.

This function returns the number of bytes generated which may be zero.

The *Z* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than (2^18)-1 = 256K bits (32385 bytes). Any error in the processing of this command is considered fatal.

Table B.23

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the *keyStream* buffer

```
LIB_EXPORT UINT16
_cpri__KDFe(
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm used in HMAC
    TPM2B           *Z,             // IN: Z
    const char      *label,         // IN: a 0 terminated label using in KDF
    TPM2B           *partyUInfo,    // IN: PartyUInfo
    TPM2B           *partyVInfo,    // IN: PartyVInfo
    UINT32           sizeInBits,    // IN: size of generated key in bits
    BYTE            *keyStream      // OUT: key buffer
    )
{
    UINT32          counter = 0;       // counter value
    UINT32          lSize = 0;
    BYTE           *stream = keyStream;
    CPRI_HASH_STATE          hashState;
    INT16           hLen = (INT16) _cpri__GetDigestSize(hashAlg);
    INT16           bytes;             // number of bytes to generate
    BYTE            marshaledUint32[4];

    pAssert(   keyStream != NULL
            && Z != NULL
            && ((sizeInBits + 7) / 8) < INT16_MAX);

    if(hLen == 0)
        return 0;

    bytes = (INT16)((sizeInBits + 7) / 8);

    // Prepare label buffer.  Calculate its size and keep the last 0 byte
    if(label != NULL)
```



```
        for(lSize = 0; label[lSize++] != 0;);

    // Generate required bytes
    //The inner loop of that KDF uses:
    //  Hashi := H(counter | Z | OtherInfo) (5)
    // Where:
    //  Hashi   the hash generated on the i-th iteration of the loop.
    //  H()     an approved hash function
    //  counter a 32-bit counter that is initialized to 1 and incremented
    //          on each iteration
    //  Z       the X coordinate of the product of a public ECC key and a
    //          different private ECC key.
    //  OtherInfo   a collection of qualifying data for the KDF defined below.
    //  In this part of ISO/IEC 11889, OtherInfo will be constructed by:
    //      OtherInfo := Use | PartyUInfo  | PartyVInfo
    for (; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
    {
        if(bytes < hLen)
            hLen = bytes;

        counter++;
        // Start hash
        if(_cpri__StartHash(hashAlg, FALSE,  &hashState) == 0)
            return 0;

        // Add counter
        UINT32_TO_BYTE_ARRAY(counter, marshaledUint32);
        _cpri__UpdateHash(&hashState, sizeof(UINT32), marshaledUint32);

        // Add Z
        if(Z != NULL)
            _cpri__UpdateHash(&hashState, Z->size, Z->buffer);

        // Add label
        if(label != NULL)
            _cpri__UpdateHash(&hashState, lSize, (BYTE *)label);
        else

            // The SP800-108 specification requires a zero between the label
            // and the context.
            _cpri__UpdateHash(&hashState, 1, (BYTE *)"");

        // Add PartyUInfo
        if(partyUInfo != NULL)
            _cpri__UpdateHash(&hashState, partyUInfo->size, partyUInfo->buffer);

        // Add PartyVInfo
        if(partyVInfo != NULL)
            _cpri__UpdateHash(&hashState, partyVInfo->size, partyVInfo->buffer);

        // Compute Hash. hLen was changed to be the smaller of bytes or hLen
        // at the start of each iteration.
        _cpri__CompleteHash(&hashState, hLen, stream);
    }

    // Mask off bits if the required bits is not a multiple of byte size
    if((sizeInBits % 8) != 0)
        keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);

    return (CRYPT_RESULT)((sizeInBits + 7) / 8);

}
```

B.9 CpriHashData.c

```
const HASH_INFO   g_hashData[HASH_COUNT + 1] = {
#if   ALG_SHA1 == YES
    {TPM_ALG_SHA1,    SHA1_DIGEST_SIZE,   SHA1_BLOCK_SIZE,
     SHA1_DER_SIZE,   SHA1_DER},
#endif
#if   ALG_SHA256 == YES
    {TPM_ALG_SHA256,    SHA256_DIGEST_SIZE,   SHA256_BLOCK_SIZE,
     SHA256_DER_SIZE,   SHA256_DER},
#endif
#if   ALG_SHA384 == YES
    {TPM_ALG_SHA384,    SHA384_DIGEST_SIZE,   SHA384_BLOCK_SIZE,
     SHA384_DER_SIZE,   SHA384_DER},
#endif
#if   ALG_SHA512 == YES
    {TPM_ALG_SHA512,    SHA512_DIGEST_SIZE,   SHA512_BLOCK_SIZE,
     SHA512_DER_SIZE,   SHA512_DER},
#endif
#if   ALG_WHIRLPOOL512 == YES
    {TPM_ALG_WHIRLPOOL512,    WHIRLPOOL512_DIGEST_SIZE,   WHIRLPOOL512_BLOCK_SIZE,
     WHIRLPOOL512_DER_SIZE,   WHIRLPOOL512_DER},
#endif
#if   ALG_SM3_256 == YES
    {TPM_ALG_SM3_256,    SM3_256_DIGEST_SIZE,   SM3_256_BLOCK_SIZE,
     SM3_256_DER_SIZE,   SM3_256_DER},
#endif
    {TPM_ALG_NULL,0,0,0,{0}}
};
```


B.10 CpriMisc.c

B.10.1. Includes

```c
#include "OsslCryptoEngine.h"
```

B.10.2. Functions

B.10.2.1. BnTo2B()

This function is used to convert a BigNum() to a byte array of the specified size. If the number is too large to fit, then 0 is returned. Otherwise, the number is converted into the low-order bytes of the provided array and the upper bytes are set to zero.

Table B.24

Return Value	Meaning
0	failure (probably fatal)
1	conversion successful

```c
BOOL
BnTo2B(
    TPM2B           *outVal,        // OUT: place for the result
    BIGNUM          *inVal,         // IN: number to convert
    UINT16           size           // IN: size of the output.
    )
{
    BYTE    *pb = outVal->buffer;

    outVal->size = size;

    size = size - (((UINT16) BN_num_bits(inVal) + 7) / 8);
    if(size < 0)
        return FALSE;
    for(;size > 0; size--)
        *pb++ = 0;
    BN_bn2bin(inVal, pb);
    return TRUE;
}
```

B.10.2.2. Copy2B()

This function copies a TPM2B structure. The compiler can't generate a copy of a TPM2B generic structure because the actual size is not known. This function performs the copy on any TPM2B pair. The size of the destination should have been checked before this call to make sure that it will hold the TPM2B being copied.

This replicates the functionality in the MemoryLib.c.

```c
void
Copy2B(
    TPM2B           *out,           // OUT: The TPM2B to receive the copy
    TPM2B           *in             // IN: the TPM2B to copy
    )
{
    BYTE        *pIn = in->buffer;
    BYTE        *pOut = out->buffer;
    int          count;
```

```
    out->size = in->size;
    for(count = in->size; count > 0; count--)
        *pOut++ = *pIn++;
    return;
}
```

B.10.2.3. BnFrom2B()

This function creates a BIGNUM from a TPM2B and fails if the conversion fails.

```
BIGNUM *
BnFrom2B(
    BIGNUM          *out,           // OUT: The BIGNUM
    const TPM2B     *in             // IN: the TPM2B to copy
    )
{
    if(BN_bin2bn(in->buffer, in->size, out) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);
    return out;
}
```


B.11 CpriSym.c

B.11.1. Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These function only use the single block encryption and decryption functions of OpesnSSL().

Currently, this module only supports AES and Camellia encryption. SM4 is not implemented in the version of OpenSSL() available to the author.

B.11.2. Includes, Defines, and Typedefs

```
#include    "OsslCryptoEngine.h"
```

SM4 is not implemented in the version of OpenSSL() available to the author

```
#ifdef TPM_ALG_SM4
#error "SM4 is not available"
#endif
typedef union {
#ifdef  TPM_ALG_AES
        AES_KEY     AesKey;
#endif
#ifdef  TPM_ALG_SM4
        SM4_KEY     SM4Key;
#endif
#ifdef  TPM_ALG_CAMELLIA
        CAMELLIA_KEY     CamelliaKey;
#endif
    } keySchedule_t;
typedef void (*encryptCall_t)(
                   const void *in,
                   void *out,
                   void *keySchedule
);
#define SET_ENCRYPT_KEY(ALG, Alg)             \
    if(0 != ALG##_set_encrypt_key(            \
                   key,                       \
                   keySizeInBits,             \
                   &keySchedule.Alg##Key)) \
        FAIL(FATAL_ERROR_INTERNAL);           \
        encrypt = (encryptCall_t)&(ALG##_encrypt)   \

#define SET_DECRYPT_KEY(ALG, Alg)             \
    if(0 != ALG##_set_decrypt_key(            \
                   key,                       \
                   keySizeInBits,             \
                   &keySchedule.Alg##Key)) \
        FAIL(FATAL_ERROR_INTERNAL);           \
        decrypt = (encryptCall_t)&(ALG##_decrypt)     \

#ifdef  TPM_ALG_AES
#   define  SET_AES_ENCRYPT     SET_ENCRYPT_KEY(AES, Aes)
#   define  SET_AES_DECRYPT     SET_DECRYPT_KEY(AES, Aes)
#else
#   define  SET_AES_ENCRYPT     pAssert(0);
#   define  SET_AES_DECRYPT     pAssert(0);
#endif
#ifdef  TPM_ALG_SM4
#   define  SET_SM4_ENCRYPT     SET_ENCRYPT_KEY(SM4, SM4)
#   define  SET_SM4_DECRYPT     SET_DECRYPT_KEY(SM4, SM4)
#else
```

```
#   define  SET_SM4_ENCRYPT     pAssert(0);
#   define  SET_SM4_DECRYPT     pAssert(0);
#endif
#ifdef  TPM_ALG_CAMELLIA
#   define  SET_CAMELLIA_ENCRYPT     SET_ENCRYPT_KEY(CAMELLIA, Camellia)
#   define  SET_CAMELLIA_DECRYPT     SET_DECRYPT_KEY(CAMELLIA, Camellia)
#else
#   define  SET_CAMELLIA_ENCRYPT    pAssert(0);
#   define  SET_CAMELLIA_DECRYPT    pAssert(0);
#endif
#define     SELECT(algorithm, direction)        \
    switch (algorithm)                          \
    {                                           \
        case ALG_AES_VALUE:                     \
            SET_AES_##direction;                \
            break;                              \
        case ALG_SM4_VALUE:                     \
            SET_SM4_##direction;                \
            break;                              \
        case ALG_CAMELLIA_VALUE:                \
            SET_CAMELLIA_##direction;           \
            break;                              \
        default:                                \
            pAssert(0);                         \
            break;                              \
    }
```

B.11.3. Utility Functions

B.11.3.1. _cpri_SymStartup()

```
LIB_EXPORT BOOL
_cpri__SymStartup(
    void
)
{
    return TRUE;
}
```

B.11.3.2. _cpri__GetSymmetricBlockSize()

This function returns the block size of the algorithm.

Table B.25

xReturn Value	Meaning
<= 0	cipher not supported
> 0	the cipher block size in bytes

```
LIB_EXPORT INT16
_cpri__GetSymmetricBlockSize(
    TPM_ALG_ID       symmetricAlg,  // IN: the symmetric algorithm
    UINT16           keySizeInBits  // IN: the key size
    )
{
    switch (symmetricAlg)
    {
#ifdef TPM_ALG_AES
    case TPM_ALG_AES:
#endif
#ifdef TPM_ALG_CAMELLIA
```



```
        // AES, Camellia and SM4 use
        // the same block size
    case TPM_ALG_CAMELLIA:
#endif
#ifdef TPM_ALG_SM4 // Both AES and SM4 use the same block size
    case TPM_ALG_SM4:
#endif
        if(keySizeInBits != 0)  // This is mostly to have a reference to
            // keySizeInBits for the compiler
            return  16;
        else
            return 0;
        break;

    default:
        return 0;
    }
}
```

B.11.4. Symmetric Encryption

```
LIB_EXPORT CRYPT_RESULT
_cpri__SymmetricEncrypt(
    BYTE               *dOut,          // OUT:
    TPM_ALG_ID          algorithm,     // IN: the symmetric algorithm
    UINT16              keySizeInBits, // IN: key size in bits
    const BYTE         *key,           // IN: key buffer. The size of this buffer
                                       //     in bytes is (keySizeInBits + 7) / 8
    TPM2B_IV           *ivInOut,       // IN/OUT: IV for decryption.
    TPM_ALG_ID          mode,          // IN: Mode to use
    UINT32              dInSize,       // IN: data size (may need to be a
                                       //     multiple of the blockSize)
    const BYTE         *dIn            // IN: data buffer
    )
{
    BYTE               *pIv;
    INT32               dSize;         // Need a signed version
    int                 i;
    BYTE                tmp[MAX_SYM_BLOCK_SIZE];
    BYTE               *pT;
    keySchedule_t       keySchedule;
    INT32               blockSize;
    encryptCall_t       encrypt;
    BYTE               *iv;

    pAssert(dOut != NULL && key != NULL && ivInOut != NULL && dIn != NULL &&
            dInSize <= INT32_MAX);
    if(dInSize == 0)
        return CRYPT_SUCCESS;

    dSize = (INT32)dInSize;
    blockSize = ivInOut->t.size;
    iv = ivInOut->t.buffer;

    // Create encrypt key schedule and set the encryption function pointer
    SELECT(algorithm, ENCRYPT);
    switch (mode)
    {
        case TPM_ALG_CTR:
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the current value of the IV(counter)
                encrypt(iv, tmp, &keySchedule);

                //increment the counter (counter is big-endian so start at end)
```

```
                for(i = blockSize-1; i >= 0; i--)
                    if((iv[i] += 1) != 0)
                        break;

                // XOR the encrypted counter value with input and put into output
                pT = tmp;
                for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = *dIn++ ^ *pT++;
            }
            break;
        case TPM_ALG_OFB:
            // This is written so that dIn and dOut may be the same
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the current value of the "IV"
                encrypt(iv, iv, &keySchedule);

                // XOR the encrypted IV into dIn to create the cipher text (dOut)
                pIv = iv;
                for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = (*pIv++ ^ *dIn++);
            }
            break;
        case TPM_ALG_CBC:
            // For CBC the data size must be an even multiple of the
            // cipher block size
            if((dSize % blockSize) != 0)
                return CRYPT_PARAMETER;
            // XOR the data block into the IV, encrypt the IV into the IV
            // and then copy the IV to the output
            for(; dSize > 0; dSize -= blockSize)
            {
                pIv = iv;
                for(i = blockSize; i > 0; i--)
                    *pIv++ ^= *dIn++;
                encrypt(iv, iv, &keySchedule);
                pIv = iv;
                for(i = blockSize; i > 0; i--)
                    *dOut++ = *pIv++;
            }
            break;
        case TPM_ALG_CFB:
            // Encrypt the IV into the IV, XOR in the data, and copy to output
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the current value of the IV
                encrypt(iv, iv, &keySchedule);
                pIv = iv;
                for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    // XOR the data into the IV to create the cipher text
                    // and put into the output
                    *dOut++ = *pIv++ ^= *dIn++;
            }
            // If the inner loop (i loop) was smaller than blockSize, then dSize would
            // have been smaller than blockSize and it is now negative. If it is
negative,
            // then it indicates how many bytes are needed to pad out the IV for
            // the next round.
            for(; dSize < 0; dSize++)
                *pIv++ = 0;
            break;

        case TPM_ALG_ECB:
            // For ECB the data size must be an even multiple of the
            // cipher block size
            if((dSize % blockSize) != 0)
```

```
                return CRYPT_PARAMETER;
            // Encrypt the inpupt block to the output block
            for(; dSize > 0; dSize -= blockSize)
            {
                encrypt(dIn, dOut, &keySchedule);
                dIn = &dIn[blockSize];
                dOut = &dOut[blockSize];
            }
            break;

        default:
            pAssert(0);
    }
    return CRYPT_SUCCESS;
}
```

B.11.4.1. _cpri__SymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Table B.26

Return Value	Meaning
CRYPT_SUCCESS	if success
CRYPT_PARAMETER	*dInSize* is not a multiple of the block size

```
LIB_EXPORT CRYPT_RESULT
_cpri__SymmetricDecrypt(
    BYTE                *dOut,          // OUT: the decrypted data
    TPM_ALG_ID           algorithm,     // IN: the symmetric algorithm
    UINT16               keySizeInBits, // IN: key size in bits
    const BYTE          *key,           // IN: key buffer. The size of this buffer
                                        //     in bytes is (keySizeInBits + 7) / 8
    TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption. The size of
                                        // this buffer is blockSize in bytes.
    TPM_ALG_ID           mode,          // IN: the decryption mode
    UINT32               dInSize,       // IN: data size (may need to be a multiple of
                                        //     the block size)
    const BYTE          *dIn            // IN: data buffer
)
{
    BYTE                *pIV;
    INT32                dSize;         // Need a signed version
    int                  i;
    BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
    BYTE                *pT;
    keySchedule_t        keySchedule;
    INT32                blockSize;
    BYTE                *iv;
    encryptCall_t        encrypt;
    encryptCall_t        decrypt;

    pAssert(dOut != NULL && key != NULL && ivInOut != NULL && dIn != NULL &&
            dInSize <= INT32_MAX);
    if(dInSize == 0)
        return CRYPT_SUCCESS;

    dSize = (INT32)dInSize;
    blockSize = ivInOut->t.size;
    iv = ivInOut->t.buffer;
    // Use the mode to select the key schedule to create.
    switch (mode)
    {
```

```
        case TPM_ALG_CBC: // decrypt = decrypt
        case TPM_ALG_ECB:
            // For ECB and CBC, the data size must be an even multiple of the
            // cipher block size
            if((dSize % blockSize) != 0)
                return CRYPT_PARAMETER;
            SELECT(algorithm, DECRYPT);
            break;
        // For these algorithms, encrypt and decrypt are the same
        case TPM_ALG_CFB:
        case TPM_ALG_CTR:
        case TPM_ALG_OFB:
            SELECT(algorithm, ENCRYPT);
            break;
    }
    // Now do the mode-dependent decryption
    switch (mode)
    {
        case TPM_ALG_CBC:
            // Copy the input data to a temp buffer, decrypt the buffer into the output;
            // XOR in the IV, and copy the temp buffer to the IV and repeat.
            for(; dSize > 0; dSize -= blockSize)
            {
                pT = tmp;
                for(i = blockSize; i > 0; i--)
                    *pT++ = *dIn++;
                decrypt(tmp, dOut, &keySchedule);
                pIv = iv;
                pT = tmp;
                for(i = blockSize; i> 0; i--)
                {
                    *dOut++ ^= *pIv;
                    *pIv++ = *pT++;
                }
            }
            break;

        case TPM_ALG_CFB:
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the IV into the temp buffer
                encrypt(iv, tmp, &keySchedule);
                pT = tmp;
                pIv = iv;
                for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    // Copy the current cipher text to IV, XOR
                    // with the temp buffer and put into the output
                    *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
            }
            // If the inner loop (i loop) was smaller than blockSize, then dSize
            // would have been smaller than blockSize and it is now negative
            // If it is negative, then it indicates how may fill bytes
            // are needed to pad out the IV for the next round.
            for(; dSize < 0; dSize++)
                *pIv++ = 0;

            break;
        case TPM_ALG_CTR:
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the current value of the IV(counter)
                encrypt(iv, tmp, &keySchedule);

                //increment the counter (counter is big-endian so start at end)
                for(i = blockSize-1; i >= 0; i--)
```



```
                    if((iv[i] += 1) != 0)
                        break;

                // XOR the encrypted counter value with input and put into output
                pT = tmp;
                for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = *dIn++ ^ *pT++;
            }
            break;

        case TPM_ALG_ECB:
            for(; dSize > 0; dSize -= blockSize)
            {
                decrypt(dIn, dOut, &keySchedule);
                dIn = &dIn[blockSize];
                dOut = &dOut[blockSize];
            }
            break;
        case TPM_ALG_OFB:
            // This is written so that dIn and dOut may be the same
            for(; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the current value of the "IV"
                encrypt(iv, iv, &keySchedule);

                // XOR the encrypted IV into dIn to create the cipher text (dOut)
                pIv = iv;
                for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = (*pIv++ ^ *dIn++);
            }
            break;
    }
    return CRYPT_SUCCESS;
}
```

B.12 RSA Files

B.12.1. CpriRSA.c

B.12.1.1. Introduction

This file contains implementation of crypto primitives for RSA. This is a simulator of a crypto engine. Vendors may replace the implementation in this file with their own library functions.

Integer format: the big integers passed in/out to the function interfaces in this library adopt the same format used in ISO/IEC 11889-1 that states:

“An integer value is considered to be an array of one or more octets. The octet at offset zero within the array is the most significant octet (MSO) of the integer."

The interface uses TPM2B as a big number format for numeric values passed to/from CryptUtil().

B.12.1.2. Includes

```
#include "OsslCryptoEngine.h"
```

B.12.1.3. Local Functions

B.12.1.3.1. RsaPrivateExponent()

This function computes the private exponent $de = 1 \bmod (p-1)*(q-1)$ The inputs are the public modulus and one of the primes.

The results are returned in the key->private structure. The size of that structure is expanded to hold the private exponent. If the computed value is smaller than the public modulus, the private exponent is de-normalized.

Table B.27

Return Value	Meaning
CRYPT_SUCCESS	private exponent computed
CRYPT_PARAMETER	prime is not half the size of the modulus, or the modulus is not evenly divisible by the prime, or no private exponent could be computed from the input parameters

```
static CRYPT_RESULT
RsaPrivateExponent(
    RSA_KEY         *key            // IN: the key to augment with the private
                                    //     exponent
    )
{
    BN_CTX          *context;
    BIGNUM          *bnD;
    BIGNUM          *bnN;
    BIGNUM          *bnP;
    BIGNUM          *bnE;
    BIGNUM          *bnPhi;
    BIGNUM          *bnQ;
    BIGNUM          *bnQr;
    UINT32           fill;

    CRYPT_RESULT     retVal = CRYPT_SUCCESS;      // Assume success

```



```
    pAssert(key != NULL && key->privateKey != NULL && key->publicKey != NULL);

    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnE = BN_CTX_get(context);
    bnD = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    bnP = BN_CTX_get(context);
    bnPhi = BN_CTX_get(context);
    bnQ = BN_CTX_get(context);
    bnQr = BN_CTX_get(context);

    if(bnQr == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);

    // Assume the size of the public key value is within range
    pAssert(key->publicKey->size <= MAX_RSA_KEY_BYTES);

    if(   BN_bin2bn(key->publicKey->buffer, key->publicKey->size, bnN) == NULL
       || BN_bin2bn(key->privateKey->buffer, key->privateKey->size, bnP) == NULL)

        FAIL(FATAL_ERROR_INTERNAL);

    // If P size is not 1/2 of n size, then this is not a valid value for this
    // implementation. This will also catch the case were P is input as zero.
    // This generates a return rather than an assert because the key being loaded
    // might be SW generated and wrong.
    if(BN_num_bits(bnP) < BN_num_bits(bnN)/2)
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
    // Get q = n/p;
    if (BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    // If there is a remainder, then this is not a valid n
    if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
    {
        retVal = CRYPT_PARAMETER;      // problem may be recoverable
        goto Cleanup;
    }
    // Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
    if(   BN_copy(bnPhi, bnN) == NULL
       || !BN_sub(bnPhi, bnPhi, bnP)
       || !BN_sub(bnPhi, bnPhi, bnQ)
       || !BN_add_word(bnPhi, 1))
        FAIL(FATAL_ERROR_INTERNAL);

    // Compute the multiplicative inverse
    BN_set_word(bnE, key->exponent);
    if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
    {
        // Going to assume that the error is caused by a bad
        // set of parameters. Specifically, an exponent that is
        // not compatible with the primes. In an implementation that
        // has better visibility to the error codes, this might be
        // refined so that failures in the library would return
        // a more informative value.  Should not assume here that
        // the error codes will remain unchanged.

        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
```

```
    fill = key->publicKey->size - BN_num_bytes(bnD);
    BN_bn2bin(bnD, &key->privateKey->buffer[fill]);
    memset(key->privateKey->buffer, 0, fill);

    // Change the size of the private key so that it is known to contain
    // a private exponent rather than a prime.
    key->privateKey->size = key->publicKey->size;

Cleanup:
    BN_CTX_end(context);
    BN_CTX_free(context);
    return retVal;
}
```

B.12.1.3.2. _cpri__TestKeyRSA()

This function computes the private exponent *de* = 1 mod (*p*-1)*(*q*-1) The inputs are the public modulus and one of the primes or two primes.

If both primes are provided, the public modulus is computed. If only one prime is provided, the second prime is computed. In either case, a private exponent is produced and placed in *d*.

If no modular inverse exists, then CRYPT_PARAMETER is returned.

Table B.28

Return Value	Meaning
CRYPT_SUCCESS	private exponent (d) was generated
CRYPT_PARAMETER	one or more parameters are invalid

```
LIB_EXPORT CRYPT_RESULT
_cpri__TestKeyRSA(
    TPM2B           *d,             // OUT: the address to receive the private
                                    //      exponent
    UINT32           exponent,      // IN: the public modulus
    TPM2B           *publicKey,     // IN/OUT: an input if only one prime is
                                    //      provided. an output if both primes are
                                    //      provided
    TPM2B           *prime1,        // IN: a first prime
    TPM2B           *prime2         // IN: an optional second prime
    )
{
    BN_CTX          *context;
    BIGNUM          *bnD;
    BIGNUM          *bnN;
    BIGNUM          *bnP;
    BIGNUM          *bnE;
    BIGNUM          *bnPhi;
    BIGNUM          *bnQ;
    BIGNUM          *bnQr;
    UINT32          fill;

    CRYPT_RESULT    retVal = CRYPT_SUCCESS;     // Assume success

    pAssert(publicKey != NULL && prime1 != NULL);
    // Make sure that the sizes are within range
    pAssert(    prime1->size <= MAX_RSA_KEY_BYTES/2
            && publicKey->size <= MAX_RSA_KEY_BYTES);
    pAssert( prime2 == NULL || prime2->size < MAX_RSA_KEY_BYTES/2);

    if(publicKey->size/2 != prime1->size)
        return CRYPT_PARAMETER;
```



```
context = BN_CTX_new();
if(context == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
BN_CTX_start(context);
bnE = BN_CTX_get(context);      // public exponent (e)
bnD = BN_CTX_get(context);      // private exponent (d)
bnN = BN_CTX_get(context);      // public modulus (n)
bnP = BN_CTX_get(context);      // prime1 (p)
bnPhi = BN_CTX_get(context);    // (p-1)(q-1)
bnQ = BN_CTX_get(context);      // prime2 (q)
bnQr = BN_CTX_get(context);     // n mod p

if(bnQr == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);

if(BN_bin2bn(prime1->buffer, prime1->size, bnP) == NULL)
    FAIL(FATAL_ERROR_INTERNAL);

// If prime2 is provided, then compute n
if(prime2 != NULL)
{
    // Two primes provided so use them to compute n
    if(BN_bin2bn(prime2->buffer, prime2->size, bnQ) == NULL)
        FAIL(FATAL_ERROR_INTERNAL);

    // Make sure that the sizes of the primes are compatible
    if(BN_num_bits(bnQ) != BN_num_bits(bnP))
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
    // Multiply the primes to get the public modulus

    if(BN_mul(bnN, bnP, bnQ, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    // if the space provided for the public modulus is large enough,
    // save the created value
    if(BN_num_bits(bnN) != (publicKey->size * 8))
    {
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
    BN_bn2bin(bnN, publicKey->buffer);
}
else
{
    // One prime provided so find the second prime by division
    BN_bin2bn(publicKey->buffer, publicKey->size, bnN);

    // Get q = n/p;
    if(BN_div(bnQ, bnQr, bnN, bnP, context) != 1)
        FAIL(FATAL_ERROR_INTERNAL);

    // If there is a remainder, then this is not a valid n
    if(BN_num_bytes(bnQr) != 0 || BN_num_bits(bnQ) != BN_num_bits(bnP))
    {
        retVal = CRYPT_PARAMETER;       // problem may be recoverable
        goto Cleanup;
    }
}
// Get compute Phi = (p - 1)(q - 1) = pq - p - q + 1 = n - p - q + 1
BN_copy(bnPhi, bnN);
BN_sub(bnPhi, bnPhi, bnP);
BN_sub(bnPhi, bnPhi, bnQ);
```



```
    BN_add_word(bnPhi, 1);
    // Compute the multiplicative inverse
    BN_set_word(bnE, exponent);
    if(BN_mod_inverse(bnD, bnE, bnPhi, context) == NULL)
    {
        // Going to assume that the error is caused by a bad set of parameters.
        // Specifically, an exponent that is not compatible with the primes.
        // In an implementation that has better visibility to the error codes,
        // this might be refined so that failures in the library would return
        // a more informative value.
        // Do not assume that the error codes will remain unchanged.
        retVal = CRYPT_PARAMETER;
        goto Cleanup;
    }
    // Return the private exponent.
    // Make sure it is normalized to have the correct size.
    d->size = publicKey->size;
    fill = d->size - BN_num_bytes(bnD);
    BN_bn2bin(bnD, &d->buffer[fill]);
    memset(d->buffer, 0, fill);
Cleanup:
    BN_CTX_end(context);
    BN_CTX_free(context);
    return retVal;
}
```

B.12.1.3.3. RSAEP()

This function performs the RSAEP operation defined in PKCS#1v2. 1. It is an exponentiation of a value (*m*) with the public exponent (*e*), modulo the public (*n*).

Table B.29

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	number to exponentiate is larger than the modulus

```
static CRYPT_RESULT
RSAEP (
    UINT32           dInOutSize,     // OUT size of the encrypted block
    BYTE            *dInOut,         // OUT: the encrypted data
    RSA_KEY         *key             // IN: the key to use
    )
{
    UINT32       e;
    BYTE         exponent[4];
    CRYPT_RESULT retVal;

    e = key->exponent;
    if(e == 0)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;
    UINT32_TO_BYTE_ARRAY(e, exponent);

    //!!! Can put check for test of RSA here

    retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut, 4, exponent,
                           key->publicKey->size, key->publicKey->buffer);

    // Exponentiation result is stored in-place, thus no space shortage is possible.
    pAssert(retVal != CRYPT_UNDERFLOW);

    return retVal;
}
```


B.12.1.3.4. RSADP()

This function performs the RSADP operation defined in PKCS#1v2. 1. It is an exponentiation of a value (*c*) with the private exponent (*d*), modulo the public modulus (*n*). The decryption is in place.

This function also checks the size of the private key. If the size indicates that only a prime value is present, the key is converted to being a private exponent.

Table B.30

Return Value	Meaning
CRYPT_SUCCESS	decryption succeeded
CRYPT_PARAMETER	the value to decrypt is larger than the modulus

```
static CRYPT_RESULT
RSADP (
    UINT32           dInOutSize,    // IN/OUT: size of decrypted data
    BYTE            *dInOut,        // IN/OUT: the decrypted data
    RSA_KEY         *key            // IN: the key
    )
{
    CRYPT_RESULT retVal;

    //!!! Can put check for RSA tested here

    // Make sure that the pointers are provided and that the private key is present
    // If the private key is present it is assumed to have been created by
    // so is presumed good _cpri__PrivateExponent
    pAssert(key != NULL && dInOut != NULL &&
            key->publicKey->size == key->publicKey->size);

    // make sure that the value to be decrypted is smaller than the modulus
    // note: this check is redundant as is also performed by _math__ModExp()
    // which is optimized for use in RSA operations
    if(_math__uComp(key->publicKey->size, key->publicKey->buffer,
                    dInOutSize, dInOut) <= 0)
        return CRYPT_PARAMETER;

    // _math__ModExp can return CRYPT_PARAMTER or CRYPT_UNDERFLOW but actual
    // underflow is not possible because everything is in the same buffer.
    retVal = _math__ModExp(dInOutSize, dInOut, dInOutSize, dInOut,
                           key->privateKey->size, key->privateKey->buffer,
                           key->publicKey->size, key->publicKey->buffer);

    // Exponentiation result is stored in-place, thus no space shortage is possible.
    pAssert(retVal != CRYPT_UNDERFLOW);

    return retVal;
}
```

B.12.1.3.5. OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus.

Table B.31

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	*hashAlg* is not valid
CRYPT_FAIL	message size is too large

```
static CRYPT_RESULT
OaepEncode(
    UINT32       paddedSize,    // IN: pad value size
    BYTE        *padded,        // OUT: the pad data
    TPM_ALG_ID   hashAlg,       // IN: algorithm to use for padding
    const char  *label,         // IN: null-terminated string (may be NULL)
    UINT32       messageSize,   // IN: the message size
    BYTE        *message        // IN: the message being padded
#ifdef  TEST_RSA                //
    ,  BYTE        *testSeed    // IN: optional seed used for testing.
#endif  // TEST_RSA             //
)
{
    UINT32       padLen;
    UINT32       dbSize;
    UINT32       i;
    BYTE         mySeed[MAX_DIGEST_SIZE];
    BYTE        *seed = mySeed;
    INT32        hLen = _cpri__GetDigestSize(hashAlg);
    BYTE         mask[MAX_RSA_KEY_BYTES];
    BYTE        *pp;
    BYTE        *pm;
    UINT32       lSize = 0;
    CRYPT_RESULT retVal = CRYPT_SUCCESS;


    pAssert(padded != NULL && message != NULL);

    // A value of zero is not allowed because the KDF can't produce a result
    // if the digest size is zero.
    if(hLen <= 0)
        return CRYPT_PARAMETER;

    // If a label is provided, get the length of the string, including the
    // terminator
    if(label != NULL)
        lSize = (UINT32)strlen(label) + 1;

    // Basic size check
    // messageSize <= k  2hLen  2
    if(messageSize > paddedSize - 2 * hLen - 2)
        return CRYPT_FAIL;

    // Hash L even if it is null
    // Offset into padded leaving room for masked seed and byte of zero
    pp = &padded[hLen + 1];
    retVal = _cpri__HashBlock(hashAlg, lSize, (BYTE *)label, hLen, pp);

    // concatenate PS of k  mLen  2hLen  2
    padLen = paddedSize - messageSize - (2 * hLen) - 2;
    memset(&pp[hLen], 0, padLen);
    pp[hLen+padLen] = 0x01;
    padLen += 1;
    memcpy(&pp[hLen+padLen], message, messageSize);

    // The total size of db = hLen + pad + mSize;
```



```
    dbSize = hLen+padLen+messageSize;

    // If testing, then use the provided seed. Otherwise, use values
    // from the RNG
#ifdef  TEST_RSA
    if(testSeed != NULL)
        seed = testSeed;
    else
#endif  // TEST_RSA
        _cpri__GenerateRandom(hLen, mySeed);

    // mask = MGF1 (seed, nSize  hLen  1)
    if((retVal = _cpri__MGF1(dbSize, mask,  hashAlg, hLen, seed)) < 0)
        return retVal; // Don't expect an error because hash size is not zero
                       // was detected in the call to _cpri__HashBlock() above.

    // Create the masked db
    pm = mask;
    for(i = dbSize; i > 0; i--)
        *pp++ ^= *pm++;
    pp = &padded[hLen + 1];

    // Run the masked data through MGF1
    if((retVal = _cpri__MGF1(hLen, &padded[1],  hashAlg, dbSize, pp)) < 0)
        return retVal; // Don't expect zero here as the only case for zero
                       // was detected in the call to _cpri__HashBlock() above.

    // Now XOR the seed to create masked seed
    pp = &padded[1];
    pm = seed;
    for(i = hLen; i > 0; i--)
        *pp++ ^= *pm++;

    // Set the first byte to zero
    *padded = 0x00;
    return CRYPT_SUCCESS;
}
```

B.12.1.3.6. OaepDecode()

This function performs OAEP padding checking. The size of the buffer to receive the recovered data. If the padding is not valid, the *dSize* size is set to zero and the function returns CRYPT_NO_RESULTS.

The *dSize* parameter is used as an input to indicate the size available in the buffer. If insufficient space is available, the size is not changed and the return code is CRYPT_FAIL.

Table B.32

Return Value	Meaning
CRYPT_SUCCESS	decode complete
CRYPT_PARAMETER	the value to decode was larger than the modulus
CRYPT_FAIL	the padding is wrong or the buffer to receive the results is too small

```
static CRYPT_RESULT
OaepDecode(
    UINT32          *dataOutSize,   // IN/OUT: the recovered data size
    BYTE            *dataOut,       // OUT: the recovered data
    TPM_ALG_ID       hashAlg,       // IN: algorithm to use for padding
    const char      *label,         // IN: null-terminated string (may be NULL)
    UINT32           paddedSize,    // IN: the size of the padded data
    BYTE            *padded         // IN: the padded data
    )
```

```
{
    UINT32        dSizeSave;
    UINT32        i;
    BYTE          seedMask[MAX_DIGEST_SIZE];
    INT32         hLen = _cpri__GetDigestSize(hashAlg);

    BYTE          mask[MAX_RSA_KEY_BYTES];
    BYTE         *pp;
    BYTE         *pm;
    UINT32        lSize = 0;
    CRYPT_RESULT retVal = CRYPT_SUCCESS;

    // Unknown hash
    pAssert(hLen > 0 && dataOutSize != NULL && dataOut != NULL && padded != NULL);

    // If there is a label, get its size including the terminating 0x00
    if(label != NULL)
        lSize = (UINT32)strlen(label) + 1;

    // Set the return size to zero so that it doesn't have to be done on each
    // failure
    dSizeSave = *dataOutSize;
    *dataOutSize = 0;

    // Strange size (anything smaller can't be an OAEP padded block)
    // Also check for no leading 0
    if(paddedSize < (<K>unsigned)((2 * hLen) + 2) || *padded != 0)
        return CRYPT_FAIL;

    // Use the hash size to determine what to put through MGF1 in order
    // to recover the seedMask
    if((retVal = _cpri__MGF1(hLen, seedMask,  hashAlg,
                        paddedSize-hLen-1, &padded[hLen+1])) < 0)
        return retVal;

    // Recover the seed into seedMask
    pp = &padded[1];
    pm = seedMask;
    for(i = hLen; i > 0; i--)
        *pm++ ^= *pp++;

    // Use the seed to generate the data mask
    if((retVal = _cpri__MGF1(paddedSize-hLen-1, mask,  hashAlg,
                        hLen, seedMask)) < 0)
        return retVal;

    // Use the mask generated from seed to recover the padded data
    pp = &padded[hLen+1];
    pm = mask;
    for(i = paddedSize-hLen-1; i > 0; i--)
        *pm++ ^= *pp++;

    // Make sure that the recovered data has the hash of the label
    // Put trial value in the seed mask
    if((retVal=_cpri__HashBlock(hashAlg, lSize,(BYTE *)label, hLen, seedMask)) < 0)
        return retVal;

    if(memcmp(seedMask, mask, hLen) != 0)
        return CRYPT_FAIL;


    // find the start of the data
    pm = &mask[hLen];
    for(i = paddedSize-(2*hLen)-1; i > 0; i--)
    {
        if(*pm++ != 0)
```



```
                break;
        }
        if(i == 0)
            return CRYPT_PARAMETER;

        // pm should be pointing at the first part of the data
        // and i is one greater than the number of bytes to move
        i--;
        if(i > dSizeSave)
        {
            // Restore dSize
            *dataOutSize = dSizeSave;
            return CRYPT_FAIL;
        }
        memcpy(dataOut, pm, i);
        *dataOutSize = i;
        return CRYPT_SUCCESS;
    }
```

B.12.1.3.7. PKSC1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1.

Table B.33

Return Value	Meaning
CRYPT_SUCCESS	data encoded
CRYPT_PARAMETER	message size is too large

```
static CRYPT_RESULT
RSAES_PKSC1v1_5Encode(
    UINT32           paddedSize,      // IN: pad value size
    BYTE            *padded,          // OUT: the pad data
    UINT32           messageSize,     // IN: the message size
    BYTE            *message          // IN: the message being padded
    )
{
    UINT32      ps = paddedSize - messageSize - 3;
    if(messageSize > paddedSize - 11)
        return CRYPT_PARAMETER;

    // move the message to the end of the buffer
    memcpy(&padded[paddedSize - messageSize], message, messageSize);

    // Set the first byte to 0x00 and the second to 0x02
    *padded = 0;
    padded[1] = 2;

    // Fill with random bytes
    _cpri__GenerateRandom(ps, &padded[2]);

    // Set the delimiter for the random field to 0
    padded[2+ps] = 0;

    // Now, the only messy part. Make sure that all the ps bytes are non-zero
    // In this implementation, use the value of the current index
    for(ps++; ps > 1; ps--)
    {
        if(padded[ps] == 0)
            padded[ps] = 0x55;    // In the < 0.5% of the cases that the random
                                  // value is 0, just pick a value to put into
                                  // the spot.
    }
```

```
    return CRYPT_SUCCESS;
}
```

B.12.1.3.8. RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2. 1.

Table B.34

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decoding error or results would no fit into provided buffer

```
static CRYPT_RESULT
RSAES_Decode(
    UINT32          *messageSize,    // IN/OUT: recovered message size
    BYTE            *message,        // OUT: the recovered message
    UINT32           codedSize,      // IN: the encoded message size
    BYTE            *coded           // IN: the encoded message
    )
{
    BOOL        fail = FALSE;
    UINT32      ps;

    fail = (codedSize < 11);
    fail |= (coded[0] != 0x00) || (coded[1] != 0x02);
    for(ps = 2; ps < codedSize; ps++)
    {
        if(coded[ps] == 0)
            break;
    }
    ps++;

    // Make sure that ps has not gone over the end and that there are at least 8
    // bytes of pad data.
    fail |= ((ps >= codedSize) || ((ps-2) < 8));
    if((*messageSize < codedSize - ps) || fail)
        return CRYPT_FAIL;

    *messageSize = codedSize - ps;
    memcpy(message, &coded[ps], codedSize - ps);
    return CRYPT_SUCCESS;
}
```

B.12.1.3.9. PssEncode()

This function creates an encoded block of data that is the size of modulus. The function uses the maximum salt size that will fit in the encoded block.

Table B.35

Return Value	Meaning
CRYPT_SUCCESS	encode successful
CRYPT_PARAMETER	hashAlg is not a supported hash algorithm

```
static CRYPT_RESULT
PssEncode   (
    UINT32        eOutSize,        // IN: size of the encode data buffer
    BYTE         *eOut,            // OUT: encoded data buffer
    TPM_ALG_ID    hashAlg,         // IN: hash algorithm to use for the encoding
```



```
    UINT32       hashInSize,     // IN: size of digest to encode
    BYTE        *hashIn          // IN: the digest
#ifdef TEST_RSA                  //
    ,  BYTE        *saltIn       // IN: optional parameter for testing
#endif // TEST_RSA               //
)
{
    INT32                   hLen = _cpri__GetDigestSize(hashAlg);
    BYTE                    salt[MAX_RSA_KEY_BYTES - 1];
    UINT16                  saltSize;
    BYTE                   *ps = salt;
    CRYPT_RESULT            retVal;
    UINT16                  mLen;
    CPRI_HASH_STATE         hashState;

    // These are fatal errors indicating bad TPM firmware
    pAssert(eOut != NULL && hLen > 0 && hashIn != NULL );

    // Get the size of the mask
    mLen = (UINT16)(eOutSize - hLen - 1);

    // Use the maximum salt size
    saltSize = mLen - 1;

//using eOut for scratch space
    // Set the first 8 bytes to zero
    memset(eOut, 0, 8);


    // Get set the salt
#ifdef  TEST_RSA
    if(saltIn != NULL)
    {
        saltSize = hLen;
        memcpy(salt, saltIn, hLen);
    }
    else
#endif  // TEST_RSA
        _cpri__GenerateRandom(saltSize, salt);

    // Create the hash of the pad || input hash || salt
    _cpri__StartHash(hashAlg, FALSE, &hashState);
    _cpri__UpdateHash(&hashState, 8, eOut);
    _cpri__UpdateHash(&hashState, hashInSize, hashIn);
    _cpri__UpdateHash(&hashState, saltSize, salt);
    _cpri__CompleteHash(&hashState, hLen, &eOut[eOutSize - hLen - 1]);

    // Create a mask
    if((retVal = _cpri__MGF1(mLen, eOut, hashAlg, hLen, &eOut[mLen])) < 0)
    {
        // Currently _cpri__MGF1 is not expected to return a CRYPT_RESULT error.
        pAssert(0);
    }
    // Since this implementation uses key sizes that are all even multiples of
    // 8, just need to make sure that the most significant bit is CLEAR
    eOut[0] &= 0x7f;

    // Before we mess up the eOut value, set the last byte to 0xbc
    eOut[eOutSize - 1] = 0xbc;

    // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
    eOut = &eOut[mLen - saltSize - 1];
    *eOut++ ^= 0x01;

    // XOR the salt data into the buffer
    for(; saltSize > 0; saltSize--)
```

```
            *eOut++ ^= *ps++;

        // and we are done
        return CRYPT_SUCCESS;
    }
```

B.12.1.3.10. PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, CRYPT_SUCCESS is returned. Any other value indicates an error.

This implementation of PSS decoding is intended for the reference TPM implementation and is not at all generalized. It is used to check signatures over hashes and assumptions are made about the sizes of values. Those assumptions are enforce by this implementation. This implementation does allow for a variable size salt value to have been used by the creator of the signature.

Table B.36

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_FAIL	decode operation failed

```
static CRYPT_RESULT
PssDecode(
    TPM_ALG_ID       hashAlg,        // IN: hash algorithm to use for the encoding
    UINT32           dInSize,        // IN: size of the digest to compare
    BYTE            *dIn,            // In: the digest to compare
    UINT32           eInSize,        // IN: size of the encoded data
    BYTE            *eIn,            // IN: the encoded data
    UINT32           saltSize        // IN: the expected size of the salt
    )
{
    INT32            hLen = _cpri__GetDigestSize(hashAlg);
    BYTE             mask[MAX_RSA_KEY_BYTES];
    BYTE            *pm = mask;
    BYTE             pad[8] = {0};
    UINT32           i;
    UINT32           mLen;
    BOOL             fail = FALSE;
    CRYPT_RESULT     retVal;
    CPRI_HASH_STATE  hashState;

    // These errors are indicative of failures due to programmer error
    pAssert(dIn != NULL && eIn != NULL);

    // check the hash scheme
    if(hLen == 0)
        return CRYPT_SCHEME;

    // most significant bit must be zero
    fail = ((eIn[0] & 0x80) != 0);

    // last byte must be 0xbc
    fail |= (eIn[eInSize - 1] != 0xbc);

    // Use the hLen bytes at the end of the buffer to generate a mask
    // Doesn't start at the end which is a flag byte
    mLen = eInSize - hLen - 1;
    if((retVal = _cpri__MGF1(mLen, mask, hashAlg, hLen, &eIn[mLen])) < 0)
        return retVal;
    if(retVal == 0)
```



```
        return CRYPT_FAIL;

    // Clear the MSO of the mask to make it consistent with the encoding.
    mask[0] &= 0x7F;

    // XOR the data into the mask to recover the salt. This sequence
    // advances eIn so that it will end up pointing to the seed data
    // which is the hash of the signature data
    for(i = mLen; i > 0; i--)
        *pm++ ^= *eIn++;

    // Find the first byte of 0x01 after a string of all 0x00
    for(pm = mask, i = mLen; i > 0; i--)
    {
        if(*pm == 0x01)
            break;
        else
            fail |= (*pm++ != 0);
    }
    fail |= (i == 0);

    // if we have failed, will continue using the entire mask as the salt value so
    // that the timing attacks will not disclose anything (I don't think that this
    // is a problem for TPM applications but, usually, we don't fail so this
    // doesn't cost anything).
    if(fail)
    {
        i = mLen;
        pm = mask;
    }
    else
    {
        pm++;
        i--;
    }
    // If the salt size was provided, then the recovered size must match
    fail |= (saltSize != 0 && i != saltSize);

    // i contains the salt size and pm points to the salt. Going to use the input
    // hash and the seed to recreate the hash in the lower portion of eIn.
    _cpri__StartHash(hashAlg, FALSE, &hashState);

    // add the pad of 8 zeros
    _cpri__UpdateHash(&hashState, 8, pad);

    // add the provided digest value
    _cpri__UpdateHash(&hashState, dInSize, dIn);

    // and the salt
    _cpri__UpdateHash(&hashState, i, pm);

    // get the result
    retVal = _cpri__CompleteHash(&hashState, MAX_DIGEST_SIZE, mask);

    // retVal will be the size of the digest or zero. If not equal to the indicated
    // digest size, then the signature doesn't match
    fail |= (retVal != hLen);
    fail |= (memcmp(mask, eIn, hLen) != 0);
    if(fail)
        return CRYPT_FAIL;
    else
        return CRYPT_SUCCESS;
}
```

B.12.1.3.11. PKSC1v1_5SignEncode()

Encode a message using PKCS1v1(). 5 method.

Table B.37

Return Value	Meaning
CRYPT_SUCCESS	encode complete
CRYPT_SCHEME	*hashAlg* is not a supported hash algorithm
CRYPT_PARAMETER	*eOutSize* is not large enough or *hInSize* does not match the digest size of *hashAlg*

```
static CRYPT_RESULT
RSASSA_Encode(
    UINT32              eOutSize,      // IN: the size of the resulting block
    BYTE               *eOut,          // OUT: the encoded block
    TPM_ALG_ID          hashAlg,       // IN: hash algorithm for PKSC1v1_5
    UINT32              hInSize,       // IN: size of hash to be signed
    BYTE               *hIn            // IN: hash buffer
    )
{
    BYTE               *der;
    INT32               derSize = _cpri__GetHashDER(hashAlg, &der);
    INT32               fillSize;

    pAssert(eOut != NULL && hIn != NULL);

    // Can't use this scheme if the algorithm doesn't have a DER string defined.
    if(derSize == 0 )
        return CRYPT_SCHEME;

    // If the digest size of 'hashAl' doesn't match the input digest size, then
    // the DER will misidentify the digest so return an error
    if((unsigned)_cpri__GetDigestSize(hashAlg) != hInSize)
        return CRYPT_PARAMETER;

    fillSize = eOutSize - derSize - hInSize - 3;

    // Make sure that this combination will fit in the provided space
    if(fillSize < 8)
        return CRYPT_PARAMETER;
    // Start filling
    *eOut++ = 0; // initial byte of zero
    *eOut++ = 1; // byte of 0x01
    for(; fillSize > 0; fillSize--)
        *eOut++ = 0xff; // bunch of 0xff
    *eOut++ = 0; // another 0
    for(; derSize > 0; derSize--)
        *eOut++ = *der++;    // copy the DER
    for(; hInSize > 0; hInSize--)
        *eOut++ = *hIn++;    // copy the hash
    return CRYPT_SUCCESS;
}
```

B.12.1.3.12. RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Table B.38

Return Value	Meaning
CRYPT_SUCCESS	decode successful
CRYPT_FAIL	decode unsuccessful
CRYPT_SCHEME	*hasIAlg* is not supported

```
static CRYPT_RESULT
RSASSA_Decode(
    TPM_ALG_ID        hashAlg,       // IN: hash algorithm to use for the encoding
    UINT32            hInSize,       // IN: size of the digest to compare
    BYTE             *hIn,           // In: the digest to compare
    UINT32            eInSize,       // IN: size of the encoded data
    BYTE             *eIn            // IN: the encoded data
    )
{
    BOOL              fail = FALSE;
    BYTE             *der;
    INT32             derSize = _cpri__GetHashDER(hashAlg, &der);
    INT32             hashSize = _cpri__GetDigestSize(hashAlg);
    INT32             fillSize;

    pAssert(hIn != NULL && eIn != NULL);

    // Can't use this scheme if the algorithm doesn't have a DER string
    // defined or if the provided hash isn't the right size
    if(derSize == 0 || (unsigned)hashSize != hInSize)
        return CRYPT_SCHEME;

    // Make sure that this combination will fit in the provided space
    // Since no data movement takes place, can just walk though this
    // and accept nearly random values. This can only be called from
    // _cpri__ValidateSignature() so eInSize is known to be in range.
    fillSize = eInSize - derSize - hashSize - 3;

    // Start checking
    fail |= (*eIn++ != 0); // initial byte of zero
    fail |= (*eIn++ != 1); // byte of 0x01
    for(; fillSize > 0; fillSize--)
        fail |= (*eIn++ != 0xff); // bunch of 0xff
    fail |= (*eIn++ != 0); // another 0
    for(; derSize > 0; derSize--)
        fail |= (*eIn++ != *der++); // match the DER
    for(; hInSize > 0; hInSize--)
        fail |= (*eIn++ != *hIn++); // match the hash
    if(fail)
        return CRYPT_FAIL;
    return CRYPT_SUCCESS;
}
```


B.12.1.4. Externally Accessible Functions

B.12.1.4.1. _cpri__RsaStartup()

Function that is called to initialize the hash service. In this implementation, this function does nothing but it is called by the CryptUtilStartup() function and must be present.

```
LIB_EXPORT BOOL
_cpri__RsaStartup(
    void
    )
```

```
{
    return TRUE;
}
```

B.12.1.4.2. _cpri__EncryptRSA()

This is the entry point for encryption using RSA. Encryption is use of the public exponent. The padding parameter determines what padding will be used.

The *cOutSize* parameter must be at least as large as the size of the key.

If the padding is RSA_PAD_NONE, *dIn* is treaded as a number. It must be lower in value than the key modulus.

NOTE If *dIn* has fewer bytes than *cOut*, then we don't add low-order zeros to *dIn* to make it the size of the RSA key for the call to RSAEP. This is because the high order bytes of *dIn* might have a numeric value that is greater than the value of the key modulus. If this had low-order zeros added, it would have a numeric value larger than the modulus even though it started out with a lower numeric value.

Table B.39

Return Value	Meaning
CRYPT_SUCCESS	encryption complete
CRYPT_PARAMETER	*cOutSize* is too small (must be the size of the modulus)
CRYPT_SCHEME	*padType* is not a supported scheme

```
LIB_EXPORT CRYPT_RESULT
_cpri__EncryptRSA(
    UINT32          *cOutSize,      // OUT: the size of the encrypted data
    BYTE            *cOut,          // OUT: the encrypted data
    RSA_KEY         *key,           // IN: the key to use for encryption
    TPM_ALG_ID       padType,       // IN: the type of padding
    UINT32           dInSize,       // IN: the amount of data to encrypt
    BYTE            *dIn,           // IN: the data to encrypt
    TPM_ALG_ID       hashAlg,       // IN: in case this is needed
    const char      *label          // IN: in case it is needed
    )
{
    CRYPT_RESULT    retVal = CRYPT_SUCCESS;

    pAssert(cOutSize != NULL);

    // All encryption schemes return the same size of data
    if(*cOutSize < key->publicKey->size)
        return CRYPT_PARAMETER;
    *cOutSize = key->publicKey->size;

    switch (padType)
    {
    case TPM_ALG_NULL:  // 'raw' encryption
        {
            // dIn can have more bytes than cOut as long as the extra bytes
            // are zero
            for(; dInSize > *cOutSize; dInSize--)
            {
                if(*dIn++ != 0)
                    return CRYPT_PARAMETER;

            }
            // If dIn is smaller than cOut, fill cOut with zeros
            if(dInSize < *cOutSize)
                memset(cOut, 0, *cOutSize - dInSize);
```



```

            // Copy the rest of the value
            memcpy(&cOut[*cOutSize-dInSize], dIn, dInSize);
            // If the size of dIn is the same as cOut dIn could be larger than
            // the modulus. If it is, then RSAEP() will catch it.
        }
        break;
    case TPM_ALG_RSAES:
        retVal = RSAES_PKSC1v1_5Encode(*cOutSize, cOut, dInSize, dIn);
        break;
    case TPM_ALG_OAEP:
        retVal = OaepEncode(*cOutSize, cOut, hashAlg, label, dInSize, dIn
#ifdef  TEST_RSA
                            ,NULL
#endif
                           );
        break;
    default:
        return CRYPT_SCHEME;
    }
    // All the schemes that do padding will come here for the encryption step
    // Check that the Encoding worked
    if(retVal != CRYPT_SUCCESS)
        return retVal;

    // Padding OK so do the encryption
    return RSAEP(*cOutSize, cOut, key);
}
```

B.12.1.4.3. _cpri__DecryptRSA()

This is the entry point for decryption using RSA. Decryption is use of the private exponent. The **padType** parameter determines what padding was used.

Table B.40

Return Value	Meaning
CRYPT_SUCCESS	successful completion
CRYPT_PARAMETER	*cInSize* is not the same as the size of the public modulus of *key*; or numeric value of the encrypted data is greater than the modulus
CRYPT_FAIL	*dOutSize* is not large enough for the result
CRYPT_SCHEME	*padType* is not supported

```
LIB_EXPORT CRYPT_RESULT
_cpri__DecryptRSA(
    UINT32            *dOutSize,      // OUT: the size of the decrypted data
    BYTE              *dOut,          // OUT: the decrypted data
    RSA_KEY           *key,           // IN: the key to use for decryption
    TPM_ALG_ID         padType,       // IN: the type of padding
    UINT32             cInSize,       // IN: the amount of data to decrypt
    BYTE              *cIn,           // IN: the data to decrypt
    TPM_ALG_ID         hashAlg,       // IN: in case this is needed for the scheme
    const char        *label          // IN: in case it is needed for the scheme
    )
{
    CRYPT_RESULT    retVal;

    // Make sure that the necessary parameters are provided
    pAssert(cIn != NULL && dOut != NULL && dOutSize != NULL && key != NULL);

    // Size is checked to make sure that the decryption works properly
```

```
    if(cInSize != key->publicKey->size)
        return CRYPT_PARAMETER;

    // For others that do padding, do the decryption in place and then
    // go handle the decoding.
    if((retVal = RSADP(cInSize, cIn, key)) != CRYPT_SUCCESS)
        return retVal;      // Decryption failed

    // Remove padding
    switch (padType)
    {
    case TPM_ALG_NULL:
        if(*dOutSize < key->publicKey->size)
            return CRYPT_FAIL;
        *dOutSize = key->publicKey->size;
        memcpy(dOut, cIn, *dOutSize);
        return CRYPT_SUCCESS;
    case TPM_ALG_RSAES:
        return RSAES_Decode(dOutSize, dOut, cInSize, cIn);
        break;
    case TPM_ALG_OAEP:
        return OaepDecode(dOutSize, dOut, hashAlg, label, cInSize, cIn);
        break;
    default:
        return CRYPT_SCHEME;
        break;
    }
}
```

B.12.1.4.4. _cpri__SignRSA()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Table B.41

Return Value	Meaning
CRYPT_SUCCESS	sign operation completed normally
CRYPT_SCHEME	*scheme* or *hashAlg* are not supported
CRYPT_PARAMETER	*hInSize* does not match *hashAlg* (for RSASSA)

```
LIB_EXPORT CRYPT_RESULT
_cpri__SignRSA(
    UINT32          *sigOutSize,    // OUT: size of signature
    BYTE            *sigOut,        // OUT: signature
    RSA_KEY         *key,           // IN: key to use
    TPM_ALG_ID       scheme,        // IN: the scheme to use
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm for PKSC1v1_5
    UINT32           hInSize,       // IN: size of digest to be signed
    BYTE            *hIn            // IN: digest buffer
    )
{
    CRYPT_RESULT    retVal;

    // Parameter checks
    pAssert(sigOutSize != NULL && sigOut != NULL && key != NULL && hIn != NULL);


    // For all signatures the size is the size of the key modulus
    *sigOutSize = key->publicKey->size;
    switch (scheme)
    {
    case TPM_ALG_NULL:
```



```
        *sigOutSize = 0;
        return CRYPT_SUCCESS;
    case TPM_ALG_RSAPSS:
        // PssEncode can return CRYPT_PARAMETER
        retVal = PssEncode(*sigOutSize, sigOut, hashAlg, hInSize, hIn
#ifdef  TEST_RSA
                            , NULL
#endif
                           );
        break;
    case TPM_ALG_RSASSA:
        // RSASSA_Encode can return CRYPT_PARAMETER or CRYPT_SCHEME
        retVal = RSASSA_Encode(*sigOutSize, sigOut, hashAlg, hInSize, hIn);
        break;
    default:
        return CRYPT_SCHEME;
    }
    if(retVal != CRYPT_SUCCESS)
        return retVal;
    // Do the encryption using the private key
    // RSADP can return CRYPT_PARAMETR
    return RSADP(*sigOutSize,sigOut, key);
}
```

B.12.1.4.5. _cpri__ValidateSignatureRSA()

This function is used to validate an RSA signature. If the signature is valid CRYPT_SUCCESS is returned. If the signature is not valid, CRYPT_FAIL is returned. Other return codes indicate either parameter problems or fatal errors.

Table B.42

Return Value	Meaning
CRYPT_SUCCESS	the signature checks
CRYPT_FAIL	the signature does not check
CRYPT_SCHEME	unsupported scheme or hash algorithm

```
LIB_EXPORT CRYPT_RESULT
_cpri__ValidateSignatureRSA(
    RSA_KEY         *key,           // IN: key to use
    TPM_ALG_ID       scheme,        // IN: the scheme to use
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm
    UINT32           hInSize,       // IN: size of digest to be checked
    BYTE            *hIn,           // IN: digest buffer
    UINT32           sigInSize,     // IN: size of signature
    BYTE            *sigIn,         // IN: signature
    UINT16           saltSize       // IN: salt size for PSS
    )
{
    CRYPT_RESULT     retVal;

    // Fatal programming errors
    pAssert(key != NULL && sigIn != NULL && hIn != NULL);

    // Errors that might be caused by calling parameters
    if(sigInSize != key->publicKey->size)
        return CRYPT_FAIL;
    // Decrypt the block
    if((retVal = RSAEP(sigInSize, sigIn, key)) != CRYPT_SUCCESS)
        return CRYPT_FAIL;
    switch (scheme)
    {
```

```
    case TPM_ALG_NULL:
        return CRYPT_SCHEME;
        break;
    case TPM_ALG_RSAPSS:
        return PssDecode(hashAlg, hInSize, hIn, sigInSize, sigIn, saltSize);
        break;
    case TPM_ALG_RSASSA:
        return RSASSA_Decode(hashAlg, hInSize, hIn, sigInSize, sigIn);
        break;
    default:
        break;
    }
    return CRYPT_SCHEME;
}
#ifndef RSA_KEY_SIEVE
```

B.12.1.4.6. _cpri__GenerateKeyRSA()

Generate an RSA key from a provided seed.

Table B.43

Return Value	Meaning
CRYPT_FAIL	exponent is not prime or is less than 3; or could not find a prime using the provided parameters
CRYPT_CANCEL	operation was canceled

```
LIB_EXPORT CRYPT_RESULT
_cpri__GenerateKeyRSA(
    TPM2B           *n,             // OUT: The public modulus
    TPM2B           *p,             // OUT: One of the prime factors of n
    UINT16           keySizeInBits, // IN: Size of the public modulus in bits
    UINT32           e,             // IN: The public exponent
    TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use in the key
                                    //     generation process
    TPM2B           *seed,          // IN: the seed to use
    const char      *label,         // IN: A label for the generation process.
    TPM2B           *extra,         // IN: Party 1 data for the KDF
    UINT32          *counter        // IN/OUT: Counter value to allow KFD iteration
                                    //     to be propagated across multiple routines
    )
{
    UINT32           lLen;          // length of the label
                                    // (counting the terminating 0);
    UINT16           digestSize = _cpri__GetDigestSize(hashAlg);

    TPM2B_HASH_BLOCK    oPadKey;

    UINT32          outer;
    UINT32          inner;
    BYTE            swapped[4];

    CRYPT_RESULT    retVal;
    int             i, fill;
    const static char     defaultLabel[] = "RSA key";
    BYTE            *pb;


    CPRI_HASH_STATE  h1;            // contains the hash of the
                                    //   HMAC key w/ iPad
    CPRI_HASH_STATE  h2;            // contains the hash of the
                                    //   HMAC key w/ oPad
    CPRI_HASH_STATE  h;             // the working hash context
```

```
    BIGNUM          *bnP;
    BIGNUM          *bnQ;
    BIGNUM          *bnT;
    BIGNUM          *bnE;
    BIGNUM          *bnN;
    BN_CTX          *context;
    UINT32           rem;

    // Make sure that hashAlg is valid hash
    pAssert(digestSize != 0);

    // if present, use externally provided counter
    if(counter != NULL)
        outer = *counter;
    else
        outer = 1;

    // Validate exponent
    UINT32_TO_BYTE_ARRAY(e, swapped);

    // Need to check that the exponent is prime and not less than 3
    if( e != 0 && (e < 3  || !_math__IsPrime(e)))
        return CRYPT_FAIL;

    // Get structures for the big number representations
    context = BN_CTX_new();
    if(context == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(context);
    bnP = BN_CTX_get(context);
    bnQ = BN_CTX_get(context);
    bnT = BN_CTX_get(context);
    bnE = BN_CTX_get(context);
    bnN = BN_CTX_get(context);
    if(bnN == NULL)
        FAIL(FATAL_ERROR_INTERNAL);

    // Set Q to zero. This is used as a flag. The prime is computed in P. When a
    // new prime is found, Q is checked to see if it is zero.  If so, P is copied
    // to Q and a new P is found.  When both P and Q are non-zero, the modulus and
    // private exponent are computed and a trial encryption/decryption is
    // performed.  If the encrypt/decrypt fails, assume that at least one of the
    // primes is composite. Since we don't know which one, set Q to zero and start
    // over and find a new pair of primes.
    BN_zero(bnQ);

    // Need to have some label
    if(label == NULL)
        label = (const char *)&defaultLabel;
    // Get the label size
    for(lLen = 0; label[lLen++] != 0;);


    // Start the hash using the seed and get the intermediate hash value
    _cpri__StartHMAC(hashAlg, FALSE, &h1, seed->size, seed->buffer, &oPadKey.b);
    _cpri__StartHash(hashAlg, FALSE, &h2);
    _cpri__UpdateHash(&h2, oPadKey.b.size, oPadKey.b.buffer);

    n->size = (keySizeInBits +7)/8;
    pAssert(n->size <= MAX_RSA_KEY_BYTES);
    p->size = n->size / 2;
    if(e == 0)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;

    BN_set_word(bnE, e);
```

```

    // The first test will increment the counter from zero.
    for(outer += 1; outer != 0; outer++)
    {
        if(_plat__IsCanceled())
        {
            retVal = CRYPT_CANCEL;
            goto Cleanup;
        }

        // Need to fill in the candidate with the hash
        fill = digestSize;
        pb = p->buffer;

        // Reset the inner counter
        inner = 0;
        for(i = p->size; i > 0; i -= digestSize)
        {
            inner++;
            // Initialize the HMAC with saved state
            _cpri__CopyHashState(&h, &h1);

            // Hash the inner counter (the one that changes on each HMAC iteration)
            UINT32_TO_BYTE_ARRAY(inner, swapped);
            _cpri__UpdateHash(&h, 4, swapped);
            _cpri__UpdateHash(&h, lLen, (BYTE *)label);

            // Is there any party 1 data
            if(extra != NULL)
                _cpri__UpdateHash(&h, extra->size, extra->buffer);

            // Include the outer counter (the one that changes on each prime
            // prime candidate generation
            UINT32_TO_BYTE_ARRAY(outer, swapped);
            _cpri__UpdateHash(&h, 4, swapped);
            _cpri__UpdateHash(&h, 2, (BYTE *)&keySizeInBits);
            if(i < fill)
                fill = i;
            _cpri__CompleteHash(&h, fill, pb);

            // Restart the oPad hash
            _cpri__CopyHashState(&h, &h2);

            // Add the last hashed data
            _cpri__UpdateHash(&h, fill, pb);

            // gives a completed HMAC
            _cpri__CompleteHash(&h, fill, pb);
            pb += fill;
        }
        // Set the Most significant 2 bits and the low bit of the candidate
        p->buffer[0] |= 0xC0;
        p->buffer[p->size - 1] |= 1;

        // Convert the candidate to a BN
        BN_bin2bn(p->buffer, p->size, bnP);

        // If this is the second prime, make sure that it differs from the
        // first prime by at least 2^100
        if(!BN_is_zero(bnQ))
        {
            // bnQ is non-zero if we already found it
            if(BN_ucmp(bnP, bnQ) < 0)
                BN_sub(bnT, bnQ, bnP);
            else
                BN_sub(bnT, bnP, bnQ);
```



```
            if(BN_num_bits(bnT) < 100)  <Q>// Difference has to be at least 100 bits
                continue;
        }
        // Make sure that the prime candidate (p) is not divisible by the exponent
        // and that (p-1) is not divisible by the exponent
        // Get the remainder after dividing by the modulus
        rem = BN_mod_word(bnP, e);
        if(rem == 0) // evenly divisible so add two keeping the number odd and
            // making sure that 1 != p mod e
            BN_add_word(bnP, 2);
        else if(rem == 1) // leaves a remainder of 1 so subtract two keeping the
            // number odd and making (e-1) = p mod e
            BN_sub_word(bnP, 2);

        // Have a candidate, check for primality
        if((retVal = (CRYPT_RESULT)BN_is_prime_ex(bnP,
                     BN_prime_checks, NULL, NULL)) < 0)
            FAIL(FATAL_ERROR_INTERNAL);

        if(retVal != 1)
            continue;

        // Found a prime, is this the first or second.
        if(BN_is_zero(bnQ))
        {
            // copy p to q and compute another prime in p
            BN_copy(bnQ, bnP);
            continue;
        }
        //Form the public modulus
        BN_mul(bnN, bnP, bnQ, context);
        if(BN_num_bits(bnN) != keySizeInBits)
            FAIL(FATAL_ERROR_INTERNAL);

        // Save the public modulus
        BnTo2B(n, bnN, n->size);  // Will pad the buffer to the correct size
        pAssert((n->buffer[0] & 0x80) != 0);

        // And one prime
        BnTo2B(p, bnP, p->size);
        pAssert((p->buffer[0] & 0x80) != 0);

        // Finish by making sure that we can form the modular inverse of PHI
        // with respect to the public exponent
        // Compute PHI = (p - 1)(q - 1) = n - p - q + 1
        // Make sure that we can form the modular inverse
        BN_sub(bnT, bnN, bnP);
        BN_sub(bnT, bnT, bnQ);
        BN_add_word(bnT, 1);

        // find d such that (Phi * d) mod e ==1
        // If there isn't then we are broken because we took the step
        // of making sure that the prime != 1 mod e so the modular inverse
        // must exist
        if(BN_mod_inverse(bnT, bnE, bnT, context) == NULL || BN_is_zero(bnT))
            FAIL(FATAL_ERROR_INTERNAL);

        // And, finally, do a trial encryption decryption
        {
            TPM2B_TYPE(RSA_KEY, MAX_RSA_KEY_BYTES);
            TPM2B_RSA_KEY        r;
            r.t.size = sizeof(n->size);

            // If we are using a seed, then results must be reproducible on each
            // call. Otherwise, just get a random number
            if(seed == NULL)
```

```
                _cpri__GenerateRandom(n->size, r.t.buffer);
            else
            {
                // this this version does not have a deterministic RNG, XOR the
                // public key and private exponent to get a deterministic value
                // for testing.
                int         i;

                // Generate a random-ish number starting with the public modulus
                // XORed with the MSO of the seed
                for(i = 0; i < n->size; i++)
                    r.t.buffer[i] = n->buffer[i] ^ seed->buffer[0];
            }
            // Make sure that the number is smaller than the public modulus
            r.t.buffer[0] &= 0x7F;
                  // Convert
            if(   BN_bin2bn(r.t.buffer, r.t.size, bnP) == NULL
                  // Encrypt with the public exponent
               || BN_mod_exp(bnQ, bnP, bnE, bnN, context) != 1
                  // Decrypt with the private exponent
               || BN_mod_exp(bnQ, bnQ, bnT, bnN, context) != 1)
                FAIL(FATAL_ERROR_INTERNAL);
            // If the starting and ending values are not the same, start over )-;
            if(BN_ucmp(bnP, bnQ) != 0)
            {
                BN_zero(bnQ);
                continue;
            }
        }
        retVal = CRYPT_SUCCESS;
        goto Cleanup;
    }
    retVal = CRYPT_FAIL;


Cleanup:
    // Close out the hash sessions
    _cpri__CompleteHash(&h2, 0, NULL);
    _cpri__CompleteHash(&h1, 0, NULL);

    // Free up allocated BN values
    BN_CTX_end(context);
    BN_CTX_free(context);
    if(counter != NULL)
        *counter = outer;
    return retVal;
}
#endif      // RSA_KEY_SIEVE
```


B.12.2. Alternative RSA Key Generation

B.12.2.1. Introduction

The files in Annex B.12.2 implement an alternative RSA key generation method that is about an order of magnitude faster than the regular method in B.12.1 and is provided simply to speed testing of the test functions. The method implemented in Annex B.12.2 uses a sieve rather than choosing prime candidates at random and testing for primeness. In this alternative, the sieve filed starting address is chosen at random and a sieve operation is performed on the field using small prime values. After sieving, the bits representing values that are not divisible by the small primes tested, will be checked in a pseudo-random order until a prime is found.

The size of the sieve field is tunable as is the value indicating the number of primes that should be checked. As the size of the prime increases, the density of primes is reduced so the size of the sieve field should be increased to improve the probability that the field will contain at least one prime. In addition, as the sieve field increases the number of small primes that should be checked increases. Eliminating a number from consideration by using division is considerably faster than eliminating the number with a Miller-Rabin test.

B.12.2.2. RSAKeySieve.h

This header file is used to for parameterization of the Sieve and RNG used by the RSA module

```
#ifndef     RSA_H
#define     RSA_H
```

This value is used to set the size of the table that is searched by the prime iterator. This is used during the generation of different primes. The smaller tables are used when generating smaller primes.

```
extern const UINT16   primeTableBytes;
```

The following define determines how large the prime number difference table will be defined. The value of 13 will allocate the maximum size table which allows generation of the first 6542 primes which is all the primes less than 2^16.

```
#define PRIME_DIFF_TABLE_512_BYTE_PAGES     13
```

This set of macros used the value above to set the table size.

```
#ifndef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   define PRIME_DIFF_TABLE_512_BYTE_PAGES    4
#endif
#ifdef PRIME_DIFF_TABLE_512_BYTE_PAGES
#   if PRIME_DIFF_TABLE_512_BYTE_PAGES > 12
#       define PRIME_DIFF_TABLE_BYTES 6542
#   else
#       if  PRIME_DIFF_TABLE_512_BYTE_PAGES <= 0
#           define PRIME_DIFF_TABLE_BYTES 512
#       else
#           define PRIME_DIFF_TABLE_BYTES (PRIME_DIFF_TABLE_512_BYTE_PAGES * 512)
#       endif
#   endif
#endif
extern const BYTE primeDiffTable [PRIME_DIFF_TABLE_BYTES];
```

This determines the number of bits in the sieve field This must be a power of two.

```
#define FIELD_POWER     14  // This is the only value in this group that should be
                            // changed
#define FIELD_BITS      (1 << FIELD_POWER)
#define MAX_FIELD_SIZE      ((FIELD_BITS / 8) + 1)
```

This is the pre-sieved table. It already has the bits for multiples of 3, 5, and 7 cleared.

```
#define SEED_VALUES_SIZE            105
const extern BYTE                   seedValues[SEED_VALUES_SIZE];
```

This allows determination of the number of bits that are set in a byte without having to count them individually.

```
const extern BYTE                   bitsInByte[256];
```

This is the iterator structure for accessing the compressed prime number table. The expectation is that values will need to be accesses sequentially. This tries to save some data access.

```
typedef struct {
    UINT32      lastPrime;
    UINT32      index;
    UINT32      final;
} PRIME_ITERATOR;
#ifdef  RSA_INSTRUMENT
#   define INSTRUMENT_SET(a, b) ((a) = (b))
#   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
#   define INSTRUMENT_INC(a)    (a) = (a) + 1
extern UINT32  failedAtIteration[10];
extern UINT32  MillerRabinTrials;
extern UINT32  totalFieldsSieved;
extern UINT32  emptyFieldsSieved;
extern UINT32  noPrimeFields;
extern UINT32  primesChecked;
extern UINT16   lastSievePrime;
#else
#   define INSTRUMENT_SET(a, b)
#   define INSTRUMENT_ADD(a, b)
#   define INSTRUMENT_INC(a)
#endif
#ifdef RSA_DEBUG
extern UINT16   defaultFieldSize;
#define NUM_PRIMES          2047
extern const __int16        primes[NUM_PRIMES];
#else
#define defaultFieldSize    MAX_FIELD_SIZE
#endif
#endif
```


B.12.2.3. RSAKeySieve.c

B.12.2.3.1. Includes and Defines

```
#include     "OsslCryptoEngine.h"
```

This file produces no code unless the compile switch is set to cause it to generate code.

```
#ifdef       RSA_KEY_SIEVE              //%
#include     "RsaKeySieve.h"
```

This next line will show up in the header file for this code. It will make the local functions public when debugging.

```
//%#ifdef    RSA_DEBUG
```

B.12.2.3.2. Bit Manipulation Functions

Introduction

These functions operate on a bit array. A bit array is an array of bytes with the 0th byte being the byte with the lowest memory address. Within the byte, bit 0 is the least significant bit.

ClearBit()

This function will CLEAR a bit in a bit array.

```
void
ClearBit(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to CLEAR
    )
{
    a[i >> 3] &= 0xff ^ (1 << (i & 7));
}
```

SetBit()

Function to SET a bit in a bit array.

```
void
SetBit(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to SET
    )
{
    a[i >> 3] |= (1 << (i & 7));
}
```

IsBitSet()

Function to test if a bit in a bit array is SET.

Table B.44

Return Value	Meaning
0	bit is CLEAR
1	bit is SET

```
UINT32
```

```
IsBitSet(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of the bit to test
    )
{
    return ((a[i >> 3] & (1 << (i & 7))) != 0);
}
```

BitsInArry()

This function counts the number of bits set in an array of bytes.

```
int
BitsInArray(
    unsigned char   *a,             // IN: A pointer to an array of bytes
    int              i              // IN: the number of bytes to sum
    )
{
    int     j = 0;
    for(; i ; i--)
        j += bitsInByte[*a++];
    return j;
}
```

FindNthSetBit()

This function finds the nth SET bit in a bit array. The caller should check that the offset of the returned value is not out of range. If called when the array does not have n bits set, it will return a fatal error

```
UINT32
FindNthSetBit(
    const UINT16     aSize,         // IN: the size of the array to check
    const BYTE      *a,             // IN: the array to check
    const UINT32     n              // IN, the number of the SET bit
    )
{
    UINT32       i;
    const BYTE  *pA = a;
    UINT32       retValue;
    BYTE         sel;

    (aSize);

    //find the bit
    for(i = 0; i < n; i += bitsInByte[*pA++]);

    // The chosen bit is in the byte that was just accessed
    // Compute the offset to the start of that byte
    pA--;
    retValue = (UINT32)(pA - a) * 8;

    // Subtract the bits in the last byte added.
    i -= bitsInByte[*pA];

    // Now process the byte, one bit at a time.
    for(sel = *pA; sel != 0 ; sel = sel >> 1)
    {
        if(sel & 1)
        {
            i += 1;
            if(i == n)
                return retValue;
        }
        retValue += 1;
    }
```



```
    FAIL(FATAL_ERROR_INTERNAL);
}
```

B.12.2.3.3. Miscellaneous Functions

RandomForRsa()

This function uses a special form of KDFa() to produces a pseudo random sequence. It's input is a structure that contains pointers to a pre-computed set of hash contexts that are set up for the HMAC computations using the seed.

This function will test that ktx. outer will not wrap to zero if incremented. If so, the function returns FALSE. Otherwise, the ktx. outer is incremented before each number is generated.

```
void
RandomForRsa(
    KDFa_CONTEXT    *ktx,           // IN: a context for the KDF
    const char      *label,         // IN: a use qualifying label
    TPM2B           *p              // OUT: the pseudo random result
    )
{
    INT16                   i;
    UINT32                  inner;
    BYTE                    swapped[4];
    UINT16                  fill;
    BYTE                    *pb;
    UINT16                  lLen = 0;
    UINT16                  digestSize = _cpri__GetDigestSize(ktx->hashAlg);
    CPRI_HASH_STATE         h;      // the working hash context

    if(label != NULL)
        for(lLen = 0; label[lLen++];);
    fill = digestSize;
    pb = p->buffer;
    inner = 0;
    *(ktx->outer) += 1;
    for(i = p->size; i > 0; i -= digestSize)
    {
        inner++;

        // Initialize the HMAC with saved state
        _cpri__CopyHashState(&h, &(ktx->iPadCtx));

        // Hash the inner counter (the one that changes on each HMAC iteration)
        UINT32_TO_BYTE_ARRAY(inner, swapped);
        _cpri__UpdateHash(&h, 4, swapped);
        if(lLen != 0)
            _cpri__UpdateHash(&h, lLen, (BYTE *)label);

        // Is there any party 1 data
        if(ktx->extra != NULL)
            _cpri__UpdateHash(&h, ktx->extra->size, ktx->extra->buffer);

        // Include the outer counter (the one that changes on each prime
        // prime candidate generation
        UINT32_TO_BYTE_ARRAY(*(ktx->outer), swapped);
        _cpri__UpdateHash(&h, 4, swapped);
        _cpri__UpdateHash(&h, 2, (BYTE *)&ktx->keySizeInBits);
        if(i < fill)
            fill = i;
        _cpri__CompleteHash(&h, fill, pb);

        // Restart the oPad hash
        _cpri__CopyHashState(&h, &(ktx->oPadCtx));
```

```

        // Add the last hashed data
        _cpri__UpdateHash(&h, fill, pb);

        // gives a completed HMAC
        _cpri__CompleteHash(&h, fill, pb);
        pb += fill;
    }
    return;
}
```

MillerRabinRounds()

Function returns the number of MillerRabin() rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```
UINT32
MillerRabinRounds(
    UINT32          bits            // IN: Number of bits in the RSA prime
    )
{
    if(bits < 511) <K>return 8;    // don't really expect this
    if(bits < 1536) <K>return 5;   // for 512 and 1K primes
    return 4;                   // for 3K public modulus and greater
}
```

MillerRabin()

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. I all likelihood, if the number is not prime, the first test fails.

If a KDFa(), PRNG context is provide (ktx), then it is used to provide the random values. Otherwise, the random numbers are retrieved from the random number generator.

Table B.45

Return Value	Meaning
TRUE	probably prime
FALSE	composite

```
BOOL
MillerRabin(
    BIGNUM          *bnW,
    int             iterations,
    KDFa_CONTEXT    *ktx,
    BN_CTX          *context
    )
{
    BIGNUM      *bnWm1;
    BIGNUM      *bnM;
    BIGNUM      *bnB;
    BIGNUM      *bnZ;
    BOOL         ret = FALSE;   // Assumed composite for easy exit
    TPM2B_TYPE(MAX_PRIME, MAX_RSA_KEY_BYTES/2);
    TPM2B_MAX_PRIME    b;
    int          a;
    int          j;
    int          wLen;
    int          i;

    pAssert(BN_is_bit_set(bnW, 0));
    INSTRUMENT_INC(MillerRabinTrials);  // Instrumentation

```


